



NATIONAL COMPUTER SECURITY CENTER

TRUSTED UNIX WORKING GROUP (TRUSIX) RATIONALE FOR SELECTING ACCESS CONTROL LIST FEATURES FOR THE UNIX® SYSTEM

18 August 1989

**Approved for Public Release:
Distribution Unlimited.**

20010802 079



NATIONAL COMPUTER SECURITY CENTER

**TRUSTED UNIX WORKING GROUP (TRUSIX)
RATIONALE FOR SELECTING
ACCESS CONTROL LIST FEATURES
FOR THE UNIX® SYSTEM**

18 August 1989

**Approved for Public Release:
Distribution Unlimited.**

20010802 079

FOREWORD

The National Computer Security Center (NCSC) formed the Trusted UNIX Working Group (TRUSIX) in 1987 to provide technical guidance to vendors and evaluators involved in the development of Trusted Computer System Evaluation Criteria (TCSEC) class B3 trusted UNIX* systems. The NCSC specifically targeted the UNIX operating system for this guidance because of its growing popularity among the government and vendor communities. By addressing the class B3 issues, the NCSC believes that this information will also help vendors understand how evaluation interpretations will be made at the levels of trust below this class. TRUSIX is making no attempt to address the entire spectrum of technical problems associated with the development of division B systems; rather, the intent is to provide examples of implementations of those security features discernible at the user interface that will be acceptable at this level of trust.

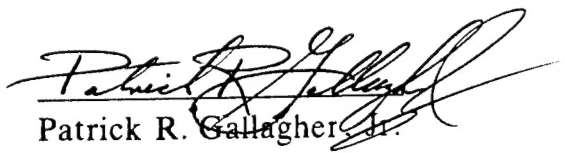
TRUSIX is not intended to be a standards body, nor does it intend to produce a de facto standard to compete against POSIX. Additionally, the TRUSIX documents are not to be construed as supplementary requirements to the TCSEC. The TCSEC is the only metric against which the trustworthiness of an operating system will be evaluated.

This document, "Rationale for Selecting Access Control List (ACL) Features for the UNIX System," is the first in a series of companion documents being produced by TRUSIX. The guidelines described in this document provide alternative methods for implementing ACLs in the UNIX system.

* UNIX is a registered trademark of AT&T

Recommendations for revision to this guideline are encouraged and will be reviewed periodically by the NCSC. Address all proposals for revision through appropriate channels to:

National Computer Security Center
9800 Savage Road
Fort George G. Meade, MD 20755-6000
Attention: Chief, Technical Guidelines Division


Patrick R. Gallagher, Jr.
Director
National Computer Security Center

18 August 1989

ACKNOWLEDGMENTS

Special recognition is extended to those members of the TRUSIX Working Group who participated in the Access Control List Subcommittee. Members of this subcommittee were: Craig Rubin, AT&T Bell Laboratories (Co-Chair); Holly Traxler, National Computer Security Center (NCSC)/Institute for Defense Analyses (IDA) (Co-Chair); Bruce Calkins, NCSC; and Casey Schaufler, Sun Microsystems. Recognition is also extended to the following members of TRUSIX who provided input through discussion and comments: Bernie Badger, Harris Corporation; Caralyn Crescenzi, NCSC; Cynthia Irvine, Gemini Computers; Howard Israel, AT&T Bell Laboratories; Frank Knowles, MITRE; James Menendez, NCSC; Dr. Eric Roskos, IDA; Rick Siebenaler, NCSC; Lucy Stasiak, AT&T Bell Laboratories; Albert Tao, Gemini Computers; Dr. Charles Testa, Infosystems Technology, Incorporated (ITI); Mario Tinto, NCSC; Grant Wagner, NCSC; Larry Wehr, AT&T Bell Laboratories; and Bruce D. Wilner, ITI.

Acknowledgment is also extended to the members of the POSIX P1003.6 Security Subcommittee and to those members of the computer security community who contributed their time and expertise by actively participating in the review of this document.

EXECUTIVE SUMMARY

The Trusted UNIX Working Group (TRUSIX) has examined the issues surrounding implementation of access control lists (ACLs) in the UNIX System and has identified a set of recommendations for implementors of ACL features. These recommendations balance issues of compatibility with existing applications, ease of use and acceptability to the end user, and architectural simplicity with the requirements for systems evaluated according to the Trusted Computer System Evaluation Criteria (TCSEC). The recommendations reflect the collected opinions and analyses of the participating vendors, evaluators, and researchers regarding implementation of ACL features.

The recommendations of TRUSIX with regard to ACLs are as follows:

- ACLs are required for files, IPC objects, and UNIX system domain sockets. Access control for sockets that use name spaces other than those local to the UNIX system (UDP, TCP) must be addressed in the specification and evaluation of the system involved, and are neither explicitly recommended nor exempted.
- Access modes specifiable via ACLs should include read, write, and execute; other modes should be allowed to be added as desired, but no additional modes should be required to be supported.
- Each ACL entry should specify permissions for either a user or a group, but not both.
- Permissions granted by an ACL entry are masked by the group class file permission bits.
- Multiple concurrent groups should be supported. In addition, some method of group subsetting should be provided. It is recommended that this subsetting allow the user to become a member of only one group at login time, then to dynamically add groups to or delete groups from the working group set as required.
- A system-defined ordering of ACL evaluation that evaluates from most specific to least specific is recommended. Where multiple concurrent groups are in use, and more than one matching group is found in the ACL, permissions granted by all matching groups should be ORed together.
- Modifications to mechanisms that change ownership, change the file permission bits, or access object attributes are not recommended.
- Existing mechanisms for object access, inquiry, and deletion should not be changed, and new parameters should not be added. Instead, new mechanisms

should be created that make use of existing ones. The interface for mechanisms that create objects should not be changed, except for the possible creation of a default ACL.

- For the new mechanisms that are added to support ACL operations, get/set operations should be used. These operations should be implemented via a single system call with command arguments to specify the various operations. For commands at the user interface, the names **getacl** and **setacl** are recommended.
- Named ACLs need not be supported.
- Provision of default ACLs for file system objects is recommended, along with a user-specifiable mechanism for indicating whether or not they should be used.
- Provision of default ACLs for IPC objects is not recommended.
- Default ACLs should be provided on a per-directory basis. Newly-created subdirectories should inherit the default ACL of the parent directory.
- When a new object is created and ACL entries are attached via a default ACL, the file group-class permission bits are not affected unless an explicit mechanism is provided.

The preceding list summarizes the recommendations of the Trusted UNIX Working Group. The main body of this document discusses the rationale for these recommendations and gives further details of the recommendations themselves. The appendix, the TRUSIX ACL Worked Example, gives an example of how these recommendations might be implemented.

CONTENTS

FOREWORD	i
ACKNOWLEDGMENTS	iii
EXECUTIVE SUMMARY	iv
CONTENTS	vi
1. Introduction	1
2. Goals	2
3. ACLs On Objects	4
3.1 ACLs On IPC Objects	4
3.2 ACLs On Sockets	5
4. Additional Access Modes	5
4.1 Require Additional Access Modes	5
4.2 Prohibit Additional Access Modes	6
4.3 Allow Additional Access Modes (With Control)	6
4.4 Allow Additional Access Modes (Without Control)	6
4.5 Recommendation	6
5. ACL Entry Type And Format	6
5.1 User And Group Entries	6
5.2 User Or Group Entries	8
5.3 Recommendation	9
6. Relationship Of ACL And File Permission Bits	10
6.1 ACL Always Replaces File Permission Bits (Pure ACL)	10
6.2 Owner Selects ACL Or File Permission Bits	11
6.3 Independent ACL And File Permission Bits (AND)	12
6.4 Independent ACL And File Permission Bits (OR)	13
6.5 File Permission Bits Contained Within ACL	13
6.6 ACL Masked By File Permission Bits	15
6.7 Recommendation	15
7. Group Semantics	16
7.1 Single Group Membership	17
7.2 Multiple Concurrent Group Membership	17
7.3 Multiple Concurrent Groups With Subsetting	18
7.4 Recommendation	19

8. ACL Evaluation	19
8.1 Ordering Of Classes	19
8.2 User-Defined Ordering	20
8.3 System-Defined Ordering	20
8.4 Multiple Group Evaluation	21
8.5 Recommendation	21
9. DAC Compatibility	22
9.1 Changing Ownership Of An Object	22
9.2 Changing The File Permission Bits	22
9.3 Creating Objects	23
9.4 Accessing Object Attributes	24
9.5 Accessing Object Data	24
9.6 Recommendation	25
10. ACL System Calls And Commands	25
10.1 Recommendation	26
11. Named ACLs	26
11.1 Recommendation	27
12. Default ACLs	27
12.1 No Default ACLs	27
12.2 Require Default ACLs	27
12.3 Provide Default ACLs	28
12.4 Recommendation	28
13. Location Of Default ACLs	28
13.1 System Wide	28
13.2 Per Process	28
13.3 Per GID Of Created File	29
13.4 Per Directory	29
13.5 Recommendation	29
14. Interaction Of Default ACL Entries At File Creation	29
14.1 OR File Group Class Permission Bits	30
14.2 AND File Group Class Permission Bits	30
14.3 No Change To File Group Class Permission Bits	30
14.4 Recommendation	30
15. Summary	31
APPENDIX: Worked Example	32
A.1 Introduction and Overview	32

A.1.1	Discretionary Access Control	32
A.1.2	Use of Access Control Lists	34
A.1.3	Structure of Access Control Lists	35
A.1.4	Discretionary Access Check Algorithm	37
A.1.5	File Object Creation	39
A.1.6	IPC Object Creation	41
A.1.7	Compatibility Requirements	41
A.1.8	Documentation Requirements	42
A.2	Commands and Functions	43
A.2.1	<i>setacl</i> Command	43
A.2.2	<i>getacl</i> Command	52
A.2.3	<i>acl</i> Function	56
A.2.4	<i>aclsort</i> Function	60
A.2.5	<i>chmod</i> Function	62
A.2.6	<i>chown</i> Function	63
A.2.7	<i>aclipc</i> Function	67
A.2.8	<i>shmctl</i> , <i>semctl</i> , & <i>msgctl</i> Functions	71
REFERENCES	72

TRUSIX Task Force: Rationale For Selecting Access Control List Features For The UNIX® System

1. Introduction

The intent of this document is to explore the issues involved in extending the UNIX System discretionary access control (DAC) mechanism. DAC is a means of controlling access to an object based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that they are chosen by the object owner.

The DAC mechanism employed in the current UNIX System was designed for efficiency, flexibility, and ease of use. This mechanism allows and encourages the sharing of information, but at a very coarse granularity, via the use of file permission bits. File permission bits are associated with three classes: owner (sometimes referred to as "user"), group, and other. Access for each class is represented by a three-bit field allowing for read, write, and execute permissions.

Several methods exist for allowing discretionary access control on objects. These methods include capabilities, profiles, access control lists (ACLs), protection bits, and password DAC mechanisms. The intent was to select a DAC mechanism with finer granularity than the current file permission bits, while maximizing the compatibility with both the current mechanism and POSIX P1003.1. Review of the methods described in *A Guide to Understanding Discretionary Access Control in Trusted Systems*[2], and of the desired outcome, point to the use of ACLs. It should be noted that ACLs can be considered a straightforward extension of the existing

* UNIX is a registered trademark of AT&T

UNIX system protection bits, since the protection bits may be interpreted to be a limited form of an ACL, which always contains three entries.

It has been suggested that the fine granularity of control provided by ACLs may be simulated in UNIX systems by using the group mechanism. Groups are lists of users which may be used to specify who may access a file. In the worst case, all possible combinations of users would have to be represented in order to fully implement these lists. This corresponds to $(2^M - 1)$ groups, where M is the number of bits in the group-ID. Since the number of possible combinations of users needed to implement this scheme for N users is $(2^N - 1)$, the maximum number of users which could effectively utilize such a system would be limited to the number of bits in the group-ID. This number (often 16 or 32) is an unreasonably small number for most UNIX systems and the management of the groups by users would be difficult. Also, this scheme does not allow for individual users in the lists to have different access rights. All users in the group would be forced to have the access rights given by the file group class permission bits. Some differences in access rights could be simulated by using the file other class permission bits, but not with the same functionality as provided by conventional ACLs.

The DAC features explored in this rationale are based on the DAC features requested by customers, the class B3 DAC requirements described in the DoD Trusted Computer Systems Evaluation Criteria [1] (TCSEC), and the DAC mechanisms used in existing trusted systems (e.g., Multics). Based on these inputs, it has been determined that the current DAC mechanism in the UNIX System is adequate for most needs and that the only enhancement required is to allow reasonable, finer-grained control of objects. This provides the capability to share or deny access to individually specified users and/or groups and meets the class B3 requirements of the TCSEC.

The issues explored in this document will deal primarily with ACLs. Much of the terminology has been adopted from the P1003.1 document and the TCSEC; however, new terms will be defined when used. For most of the issues identified, alternative solutions are given along with a recommendation. Although an attempt was made to consider the issues independently, it should be noted that some of the issues are actually very dependent on each other and recommendations made in some areas greatly influenced later recommendations.

2. Goals

The primary goal in extending discretionary access control in the UNIX system is to provide a finer granularity of control in specifying user and/or group access to objects. This can be achieved through the addition of access control lists. The

following is a list of additional goals for the extended DAC mechanism:

- The mechanism should provide compatibility with the existing (currently P1003.1) and emerging POSIX standards and with the current UNIX System DAC mechanism. In the unlikely event of a conflict between the current UNIX System DAC mechanism and POSIX, the POSIX interpretation will be used. In addition, the semantics of existing interfaces should be maintained.
- The following requirements for DAC in the TCSEC at class B3 should be fulfilled. "The TCB shall define and control access between named users and named objects (e.g., files and programs) in the ADP system. The enforcement mechanism (e.g., access control lists) shall allow users to specify and control sharing of those objects, and shall provide controls to limit propagation of access rights. The discretionary access control mechanism shall either by explicit user action or by default, provide that objects are protected from unauthorized access. These access controls shall be capable of specifying, for each named object, a list of named individuals and a list of groups of named individuals with their respective modes of access to that object. Furthermore, for each such named object, it shall be possible to specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given. Access permissions to an object by users not already possessing access permission shall only be assigned by authorized users."
- Reasonable vendor extensions to the DAC mechanism should not be precluded. For example, the specification of read, write and execute permissions should be supported. Other permissions should not be required nor should they be precluded as extensions.
- A minimum set of new interfaces and error codes should be provided. The new command interfaces provided for the user must be easy to use and the existing interfaces should continue to work as expected.
- Intermixing use of the existing and newly-defined DAC functions/commands should provide reasonable results. Security should be maximized by opting for more restrictive rather than less restrictive decisions when a choice must be made.
- When changing DAC on an object, at no time shall access be more permissive than either the initial or resulting access.

3. ACLs On Objects

A system can support several different types of objects, e.g., system objects, public objects, named objects. System objects are entities internal to the TCB (e.g., system data structures) not directly accessible by the normal user and, as such, do not require discretionary access control. Public objects are objects readable but unmodifiable to the normal user (e.g., system clock), and thus also do not require discretionary access control. Named objects are objects readable and modifiable at the user interface (e.g., text files). The TCSEC class B3 requirement for DAC states that access control must be enforced on all named objects in the system [1]. Although there may be some variance among different UNIX system implementations, there are two common classes of named objects that require ACLs. These classes are files (including regular, directory, special, and named pipes), and named IPC objects (including shared memory, message queues, semaphores, and sockets).

It is these classes of objects that will be protected by the discretionary access control alternatives described later in the paper. It should be pointed out, however, that discretionary access can not always be completely determined solely by the file permission bits and the ACL associated with the object. It is possible to have objects which have been administratively configured for a specific access and thus not completely affected by user DAC, e.g., a file system mounted read-only. There are other instances where discretionary access of objects may be time-dependent and thus not completely based on a current DAC setting. Examples of this would be the inability to write a shared-text file while it is being executed or trying to execute a file while it is open for writing. These situations are acknowledged special cases and will not be considered in the general discussion of determining effective discretionary access.

3.1 ACLs On IPC Objects

IPC objects are named objects and are thus require ACLs at class B3. Note that this does not include unnamed pipes which can only be used to connect related processes. Although the semantics of IPC mechanisms are slightly different from those of file system objects, a DAC scheme similar to that used for file system objects should easily be adaptable to IPC objects. For example, message queues utilize both a creator and an owner of an IPC object and maintain creator and owner UIDs and GIDs (cuid,uid, cgid,gid). User access is checked against the cuid and the uid, and group access is checked against the cgid and gid. This situation can easily be represented with ACLs by using additional ACL entries to represent the creator UID and GID. Additionally, some access modes associated with file system objects, such as execute, may not be applicable to IPC objects. This does not cause a

problem as long as the modes are a subset of those defined for file system objects.

3.2 ACLs On Sockets

Sockets are named objects and would thus require ACLs at class B3. UNIX system domain sockets use the file system name space for access control decisions and currently have file permission bits associated with them. Thus, domain sockets would also need to have ACLs associated with them. Other types of sockets which use other name spaces (UDP, TCP) are currently not protected with any type of access control. Since it is not clear whether these types of sockets could currently be included in an evaluated configuration, they will not be addressed at this time.

4. Additional Access Modes

Existing UNIX systems support three access modes: *read*, *write*, and *execute/search*. Additional access modes are conceivable, and may be convenient to add while adding ACLs. Various possibilities include:

- read attributes of object
- write attributes of object
- append only to object
- truncate data of object
- delete object
- lock object
- restrict setuid execution of object
- restrict access of object based on time.

Note that this is not an all-inclusive list.

In this and subsequent sections, alternative implementations of a given topic are examined, followed by the TRUSIX recommendation.

4.1 Require Additional Access Modes

In this approach to handling additional access modes, new access modes would be defined and required. This limits the availability of compliant implementations and impacts compatibility.

4.2 Prohibit Additional Access Modes

In this approach, new access modes would explicitly not be allowed. Due to loss of flexibility, compliance with this scheme would limit implementation.

4.3 Allow Additional Access Modes (With Control)

In this approach, new access modes would not be defined. Instead, the concept of and mechanism for adding new access modes would be defined. This allows a vendor to produce whatever additional access modes are desired. Since the mechanism for doing so is defined there is little chance of collisions or contradictions. The mechanisms must be defined and agreed upon by some regulating body which allocates access bits. Note no such body currently exists which has been tasked to allocate access bits.

4.4 Allow Additional Access Modes (Without Control)

In this approach, additional access modes are neither defined nor precluded. This method allows a vendor to produce whatever additional access modes are desired, but there is no mechanism provided for adding new modes. There would be no control on the access modes vendors might add.

4.5 Recommendation

We recommend allowing additional access modes, without control. There should be nothing precluding the addition of new access modes if desired. However, since there is nothing currently in the POSIX P1003.1 standard concerning additional access modes, no new access modes or mechanisms need be defined.

5. ACL Entry Type And Format

The manner in which an ACL entry refers to a user or group of users is an important factor in the usability of an ACL mechanism. The alternatives are to have an ACL entry contain either a user or group in an entry, or to have an ACL entry contain both a user and group. The issue is which of the alternatives is more suitable to a system utilizing ACLs.

5.1 User And Group Entries

A user and group entry contains a reference to both a specific user and a specific group together as a [UID,GID] pair. The UID-specific and GID-specific entries can be represented as special "wildcard" cases (denoted by *) meaning any user or group will match that entry. Using this method, an ACL entry may refer to one user in a particular group [UID,GID], one user in any group [UID,*], any user in a particular group [*,GID], or any user in any group [*,*] which is equivalent to the file other

class permission bits. A typical ACL utilizing entries of this type might look like the following:

user1.projA	rw-
user2.projB	r--
user3.*	rwX
*.projA	r--
.	---

Implementations of protected subsystems is the only clear example that suggests using user and group ACL entries as a pair. Using the UNIX system `setgid-on-exec` feature, it is possible to build protected subsystems. Consider the following example which makes use of this feature.

A database of tapes is maintained in `/etc/tapedata`. The database administrator (DBA) of the database wishes to produce a utility to control access to this database. To begin with, there are some rules for dealing with the database. Some users should have read and write access, others just read access, and still others should have no access to the database. Readers should only see data about their own tapes. In addition, since other database utilities have poor error handling, all updates to the database need to be made in the correct format.

The DBA has written a utility named `tapedb` which can read and update the database. `/etc/tapedata` and `tapedb` both have the group `tape` associated with them, and `tapedb` has the set-group-id bit on. The DBA has also created an ACL for `/etc/tapedata` which contains the following entries:

user1.tape	r--
user2.tape	r--
user3.tape	rw-
user4.tape	rw-
.	---

All users named in the ACL (in group `tape`) may read the database. Only `user3` and `user4` (in group `tape`) may update the database. If the only way for a user to be a member of group `tape` is by executing `tapedb`, then the DBA is satisfied that `/etc/tapedata` is adequately protected.

While this example suggests a useful application of user and group ACL entries, there are other ways to implement the example which do not require this ACL entry type functionality. As described in the following section, the same effect can be

achieved through ACLs containing user or group entries.

Additionally, identification by a user and group pair is not used in a UNIX System. In some systems, a user is identified by a user-ID, group-ID pair. In Multics, for example, a user is identified by a user-ID, project-ID pair, where a project-ID is equivalent to a group-ID on the UNIX system. User1 in projA, on a Multics system, is distinct from user1 in projB. Since Multics users do not have the capability to change groups, the only way for a user to be identified with another project would be to log in with another group-ID. In UNIX systems, however, a user is really only identified by the user-ID. Also, a user can easily change group-ID through the *newgrp* command or be associated with several groups at the same time if using a system with multiple groups. Thus, controlling access for a user while in a specific group is not as useful in a UNIX system.

5.2 User Or Group Entries

A user or group entry contains a reference to either a specific user or a specific group, but only one at a time. Consider the following example, where *u* indicates the user class, *g* indicates the group class, and *o* indicates the other class:

u:user1	rw-
u:user2	r--
u:user3	rw-
u:user4	rw-
g:projA	r--
g:projB	rw-
o:	rw-

To address the protected subsystem implementation, consider again the tape database example described in the previous section. Rather than controlling access to the data, access can be controlled on two subprograms; one which reads data, the other which updates data. The ACL on the database, */etc/tapedata* would be:

g:tapereaders	r--
g:tapewriters	rw-
o:	---

The user interface for access to the database is *tapedb*. The program *tapedb* is not setgid, however, it invokes two other programs, *tapedb_read* and *tapedb_write*, that are setgid. Only users allowed to read the database have execute permission on *tapedb_read*, while only those allowed to update the database may execute

tapedb_write. The ACL on *tapedb_read* would be:

u:user1	--x
u:user2	--x
u:user3	--x
u:user4	--x
o:	---

The ACL on *tapedb_write* would be:

u:user3	--x
u:user4	--x
o:	---

The program *tapedb_read* runs *setgid* to the group **tapereaders**, and the program *tapedb_write* runs *setgid* to the group **tapewriters**.

Thus, the same protected subsystem can be provided through ACLs of type user or group.

The main advantage of this scheme is that it provides more clarity for the user. This is considered to be a very important advantage as a user's understanding of such a mechanism is essential in promoting its correct usage. Additionally, this scheme removes the need for wildcard specifiers, thus eliminating the potential problems of picking an unused character as a specifier.

5.3 Recommendation

User or group entries in ACLs are recommended. Since there is no clear need for the user-group paired entry scheme and there are several advantages to the user or group scheme, the user or group scheme is the preferred alternative. Examples were examined which claimed to require the use of user-group paired entries. One such example deals with protected subsystems as described above. Protected subsystems, a useful and important feature in a trusted system, can be implemented through other means not requiring user-group paired entries. We have found that this is a limited class of applications and may be implemented with the user or group scheme with minimal effort. For UNIX systems with multiple groups, the user and group scheme becomes more difficult when determining access. Additionally, the user or group scheme follows the idea in UNIX systems that a user is only identified by user-ID and gives no special meaning to what a user can do while only in a certain group. Finally, although simplicity is a very subjective measure, in comparing the two alternatives the advantage of simplicity outweighs the ability to specify both a

user and a group in a single entry.

6. Relationship Of ACL And File Permission Bits

ACLs expand upon the discretionary access control facility which is already provided by the file permission bits. Although file permission bits do not meet the TCSEC class B3 requirement for DAC, they are sufficient for many uses and are the only mechanism available to existing applications. Existing applications that are security-conscious use file permission bits to control access. The relationship between the ACL and the file permission bits is important to existing programs in order to maintain compatibility. For example, use of `chmod("object", 0)` should continue to work, denying subsequent opens to an object. The following sections discuss possible approaches to handling the interaction of ACLs with file permission bits. Any references to default ACLs will be fully described in the *Default ACLs* section.

6.1 ACL Always Replaces File Permission Bits (Pure ACL)

In this approach, the file permission bits are no longer consulted for DAC decisions. Instead, each object always has an ACL and the ACL completely determines access.

Consider the following example illustrating this scheme. Assume User1 and User2 are members of the group "GroupA" and User3 and User4 are not.

file Owner/Group	User2/GroupA
file permission bits:	<code>rwxr-x--x</code>
ACL Entries:	
User1	<code>rwX</code>
User2	<code>r--</code>
User3	<code>rwX</code>
User4	<code>---</code>

In this example the file permission bits would have no effect on the access control decision. User3 is able to read, write and execute the file. User2 is able to read it, but not to execute or write to the file. The file permission bits are completely ignored.

The resulting pure ACL system does not have to worry about interactions between the ACL and the file permission bits, since the latter are not used for access control decisions. A single, well defined access policy is employed. Applications which should make use of DAC are forced to understand the new rules.

The major disadvantage of this scheme, however, is that compatibility is lost. Every DAC cognizant program, and that should be every program that manipulates the discretionary access control information on an object needs to be changed to understand ACLs.

6.2 Owner Selects ACL Or File Permission Bits

In this approach, either the file permission bits or the ACL are consulted for the access control decision on a per object basis. The owner determines whether the file permission bits or the ACL is used. The system call **chmod** returns an indicative error if the object has an ACL, but otherwise sets the file permission bits.

Consider the two following examples which illustrate this approach. Once again assume User1 and User2 are members of the group "GroupA" and User3 and User4 are not.

Example A (ACL selected):

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x
ACL Entries:	
User1	rwX
User2	r--
User3	rwX
User4	---

Since there is an ACL on this file the access control is the same as in the previous example.

Example B (file permission bits selected):

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x
ACL Entries:	NONE

Since there are no ACL entries on this file the access control is determined by the permission bits. User2 (owner) has all access permissions to the file. User1 (a user in GroupA) is allowed read and execute access. User3 and User4 ("other" users) can only execute the file.

The resulting system behaves like a *file permission bit based system* if no one ever sets ACLs and like the *pure ACL system* if a default ACL mechanism is in use. Thus, either environment can be supported.

The compatibility issues raised in the previous section apply here as well. In addition, the programs have to determine which access control mechanism applies to each object created and set the DAC accordingly.

6.3 Independent ACL And File Permission Bits (AND)

In this approach, both the file permission bits and the ACL are consulted for the discretionary access control decision on a per object basis. Access is granted if and only if it is granted by both the ACL and the file permission bits.

Consider the following example, which illustrates this approach. For this example, assume only User2 is in GroupA.

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x
ACL Entries:	
User1	rwx
User2	r--
User3	r-x
User4	---

In the example above, the file permission bits imply that User1 has execute permission, whereas the permissions specified in the ACL imply that User1 has full access. Without knowing which group User1 is in, one cannot predict whether or not User1 can read the file. If User1 is in group GroupA, then User1 will have read and execute permissions. If User1 is not in group GroupA, then only execute permission will be granted. Similarly, without knowing User3's group, one cannot predict whether or not User3 has read access. User4 will have no possibility of access, due to no permissions specified in the ACL entry. As the example illustrates, there is no way to get a full ACL view with this scheme.

With this scheme, some compatibility is maintained. Calls to **chmod** have the desired effect from the restrictive point of view. ACL entries can further restrict access.

Making use of the ACL as the effective access control mechanism requires that the file permission bits be set wide-open (i.e., read, write, and execute bits are set for user, group and other). In situations where ACLs are not properly set, a new object will become generally accessible. Likewise, if the ACL is removed then the object will again be generally accessible. This scheme also allows for misleading status information given to programs which only use the existing mechanism.

6.4 Independent ACL And File Permission Bits (OR)

In this approach, both the file permission bits and the ACL are consulted for the discretionary access control decision on a per object basis. Access is granted if it is granted by either the ACL or the file permission bits. The ACL is used to grant access beyond what is set in the file permission bits.

Consider the following example illustrating this approach. Assume only User2 is in GroupA.

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x
ACL Entries:	
User1	rwX
User2	r--
User3	rwX
User4	---

User1, User2, and User3 have read, write, and execute access. User4 has execute access.

Again, some compatibility is maintained. Calls to **chmod** have the desired effect from the permissive point of view. The previous alternative's problem of leaving the permission bits wide-open is thus avoided.

The problem with this scheme, however, is that a **chmod** call which would deny all access (**chmod**("object", 0)) in a system without ACLs will not do so here.

6.5 File Permission Bits Contained Within ACL

In this approach, only the ACL is consulted for discretionary access control decisions. The file permission bits are replaced by three "base" entries in the ACL. Calls to **chmod** modify the **owner**, **group**, and **other** entries contained in the ACL. Calls to **stat** read this information from the ACL.

In the following two examples assume the **owner** entry is evaluated before additional user entries, and the **group** entry is evaluated before additional group entries.

Example A:

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x

ACL Entries:

owner	rwX
User1	rwX
User2	r--
User3	r-X
User4	---
group	r-X
other	--X

In this example, it is not clear what permissions User2 is to be granted, since a particular method for determining owner access has not been specified for the case where an additional user entry also names the owner. User2 could be granted read, write, and execute access as the owner, read access only, as per the explicit entry for User2, or some combination of the two (e.g., the AND or OR of the two). User1, User3, and User4 get their access from their ACL entries.

Example B: (After a **chmod("object", 0)**)

file Owner/Group	User2/GroupA
file permission bits:	-----
ACL Entries:	
owner	---
User1	rwX
User2	r--
User3	r-X
User4	---
group	---
other	---

Changing the file permission bits to zero does not change the permissions granted to User1, User3, and User4, since their access is based on ACL entries. User2's access may change depending on how owner access is determined when additional user entries naming the owner also exist.

If no additional entries are added to the ACLs, this system looks like a system without ACLs. The literal meaning of the file permission bits is preserved in the ACL.

As in the previous alternative, however, a **chmod** call which would deny all access (**chmod("object", 0)**) in a system without ACLs will not do so here.

6.6 ACL Masked By File Permission Bits

In this approach, both the file permission bits and the ACL are used for determining the discretionary access control decision. The access indicated in the ACL entry is logically ANDed (masked) with one or more of the file permission bit classes (file owner, file group, or file other class) to determine the effective DAC permission.

Example:

file Owner/Group	User2/GroupA
file permission bits:	rwxr-x--x
ACL Entries:	
User1	rwx
User2	r--
User3	r-x
User4	---

Assume that the group file permission bits are chosen as the mask, i.e., all ACL entries will be ANDed against the file group class permission bits. User2, being the owner, gets read, write, and execute access to the file. User3 is allowed read and execute access. User1 is allowed read and execute access, the write access is disallowed by the file permission bits. User4 is not allowed any access to the file.

Calls to **chmod** have the desired effect from the restrictive point of view but not necessarily from the permissive point of view. Since the bits of the masked field will most likely be set wide-open, the literal meaning of the field chosen for the mask appears to be lost. The POSIX standard, however, allows for the extended meaning of the group class permission bits.

6.7 Recommendation

We recommend the *ACL Masked By File Permission Bits* approach. This is the most reasonable approach when trying to balance security and compatibility. The question of designating the masking field must still be resolved. The file group class permission bits are the preferred masking field, even though they encourage permissive default access by the owning group. This choice must be made because the use of the file owner class would cause compatibility problems in programs which attempt to establish "owner-only" access, whereas the designation of the file other class could leave objects open to attack were an ACL removed or never present. An additional option of masking user entries with the file owner class permission bits and group entries with the file group class permission bits has the same disadvantages as masking against only the file owner class. When masking

against the file group class, the permissions indicate the least upper bound of the permissions allowed for the ACL entries and the user and other fields retain their previous semantics.

To summarize the approaches identified in this section:

The *ACL Masked By File Permission Bits* approach is a compromise for both security and compatibility.

The *Independent ACL And File Permission Bits (AND)* approach suffers from the serious flaw that the file permission bits must be set very permissively in order to allow the ACL entries to predominate in the discretionary access calculation. A simple mistake in setting the ACL could grant object access to significantly more users than was intended.

The *Independent ACL And File Permission Bits (OR)* approach may require that both ACL and the file permission bits be changed in order to deny a particular access. Thus, existing programs could believe that they had prevented access when they, in fact, had not. Similarly, in the *File Permission Bits Contained Within ACL* approach, removing "other" permission might not have the desired effect, since, the owner, group, and other entries may not be the only ones in the ACL. In neither case does a call to **chmod** with a zero argument unequivocally revoke access from all users as might be expected.

Whichever DAC scheme is ultimately selected, an appropriate balance must be struck between the mutually conflicting concerns of compatibility and security. In a DAC scheme where **chmod** cooperates with ACLs, **chmod** must not grant inappropriate access or require unreasonable (i.e., permissive public access) defaults.

Barring compatibility, the alternatives of ACLs replacing file permission bits (Pure ACLs and On Demand) would be the most elegant way of enhancing DAC for UNIX systems. By abandoning file permission bits, however, these schemes have been rendered incompatible with existing systems. Thus, they are not considered for a POSIX-compliant UNIX system DAC scheme.

7. Group Semantics

There are various ways of using the UNIX system group mechanism when grouping system users. In designing ACLs it is important to understand the possible semantics and provide enough flexibility to properly support these semantics. Initially, there are no restrictions on how users can be grouped. Various possibilities include:

- a shorthand way of referring to groups of subjects

- a method of grouping project work by group access rights
- privileged roles
- accountability (file ownership)

The issue arises, however, of how to deal with user membership when considering these possible grouping mechanisms. For example, should a user be permitted to be a member of more than one group at any given time? If so, should there be a mechanism provided to allow the user to control group membership? These issues will be addressed in the following sections.

7.1 Single Group Membership

Under a single membership scheme, a user can only be a member of one specific group at any given time. All discretionary access checks will be made with respect to the user's UID and a single GID. A user will only be able to change his/her group through the use of the **newgrp** command. This scheme is easy to implement and introduces no additional complexity with respect to evaluating access within an ACL. Additionally, it would certainly be acceptable in a class B3 system.

7.2 Multiple Concurrent Group Membership

Under a multiple concurrent group scheme, a user can be a member of more than one group at the same time. This scheme introduces some complexity when evaluating user access by allowing more than one ACL entry of equal specificity to apply to a user simultaneously. For example, if a user is a member of several groups at the same time and tries to access an object with an ACL containing entries which match the user on more than one group, what will the resulting access be? There are several ways of determining the resulting access in such a case. These are discussed under *ACL Evaluation*.

Another concern with the use of multiple concurrent groups is the possibility of violating the least privilege principle. With multiple concurrent groups if a user is in several groups at once, he/she is granted access to all of those groups at all times rather than to just the ones he/she needs at any given time. This could be contrary to the idea of a user having a minimal set of privileges necessary to perform a particular function at any given time.

It can be argued, however, that the least privilege requirement in the TCSEC only applies to TCB architecture, making this issue irrelevant for DAC. On the other hand there may be a problem with a system which implements privileged roles through the group mechanism. The TCSEC class B3 Trusted Facility Management requirement states that separate roles must be assigned to operator and

administrator functions and that each role be restricted to performing only those functions necessary for that role. Given a system, therefore, which uses the group mechanism to assign roles and grant access based on role identity to parts of the system which would otherwise be inaccessible, it is clear that least privilege could be violated through the use of multiple concurrent groups. The violation would occur if the user who was a member of the group assigned to a privileged role could also be a member of one or more additional groups. Proper administration of these privileged groups, however, could still allow for the use of multiple groups, but a subsetting capability, as described in the next subsection, would then be required.

Improperly controlled multiple concurrent groups with groups representing privileged roles could therefore be a violation of the least privilege principle. This would result in a failure to meet the class B3 requirements. This is only one specific implementation, however, and it is certainly conceivable that multiple concurrent groups could be implemented in such a way as to not be a violation of least privilege. The multiple concurrent group scheme is currently a feature in some UNIX systems and is thought to be an extremely useful and necessary feature to those who use it. Multiple concurrent groups would also be compatible with the POSIX standard.

7.3 Multiple Concurrent Groups With Subsetting

Another problem associated with multiple concurrent groups arises from the fact that currently when a user logs on to a system he/she automatically becomes a member of all of the groups that he/she is allowed membership in. There is no way for the user to only be active in a subset of his/her possible group set. Although there is no explicit requirement in the TCSEC precluding this, the TCSEC does seem to imply that a user should by default have a minimal amount of access rights at login.

There are several ways of approaching this problem; any of these methods would be a possible and acceptable means of resolving this problem. First, it is necessary to consider whether a user should be able to add or delete groups from his/her group set and if so, with what restrictions. A user should certainly not be allowed to add groups for which he/she is not authorized. Therefore each user should have an "allowable group set" which consists of all groups that user has been given authorization to be a member of. Adding groups other than those which appear in this allowable group set would be unacceptable.

There are at least two ways to allow a user to work with a subset of his/her allowable group set. The first would be to keep the current scheme where a user becomes a member of all of his/her groups at login, but provide the user with a means (through some system call or command) to drop specific groups if desired and work as a

member of some subset of his/her allowable group set. A command would allow a user the capability but require an explicit action to do so. A system call, on the other hand, would provide the means for restriction through a program which could be set up to run automatically for the user. This would mean, however, that the set of groups would either be hardcoded into the program or be set through some type of configuration file. Another possible approach would be to provide a mechanism that would cause a program's groups to be restricted when that program is executed. Although this eliminates the user having to remember to restrict his/her groups or having to hardcode a group set into a program, it would add further complexity to the system.

7.4 Recommendation

We recommend that the multiple concurrent group capability be provided along with some method of subsetting. The preferred method would be to only allow the user to become a member of one group at login and provide him/her with a means of dynamically adding/deleting to his/her working group set. This recommendation, of course, may conflict with implementations which use the group mechanism for privilege roles.

8. ACL Evaluation

This issue deals with how an ACL is evaluated to determine access rights of a subject to a particular object. There are several possible ordering methods for ACL evaluation, as well as several different ways to evaluate multiple group entries.

Two levels of ordering must actually be considered when deriving an ACL evaluation scheme; the ordering of the classes (user, group, other), and then the ordering of the entries within each class.

8.1 Ordering Of Classes

It would certainly be possible to specify an ordering of any combination of the three classes, user, group, and other. However, since both the POSIX standard and all current UNIX systems specify a "user, then group, then other" ordering, (or most-to-least specific), when evaluating access with permission bits, this ordering should be maintained for ACLs as well.

The method of evaluating an ACL in a most-to-least specific manner can be described as follows. The owner identity of the object is first checked against the effective identity of the subject. If there is a match the search stops. Next, a check is made against the owning group identity of the object and the effective group of the subject. If there is a match and the subject does not have multiple groups, the search

stops. Otherwise the rest of the group entries are searched next. If the subject has multiple groups, the group entries are evaluated as presented in the *Multiple Group Evaluation* section, otherwise they are searched in order as the user entries are. Finally, if no user or group entries were found to match the effective identity of the subject, access is determined based on the **other** entry.

For the following discussion on the ordering of ACL entries, it will be assumed that the classes will be ordered and follow this most-to-least specific regime.

8.2 User-Defined Ordering

In this method, entries are considered according to the ordering given by the user. The first entry as specified by the user is considered first, the second entry next, and so on.

As long as the "user, then group, then other" order is followed, the only security relevant problem with this method occurs when evaluating group entries with multiple groups. If a user is a member of multiple groups and matches more than one of the group entries, the resulting access may be dependent upon the ordering of the group entries. See the *Multiple Group Evaluation* section for various possibilities. Unless all matching group entries are considered when determining access, the burden is placed on the user to correctly order the group entries.

This method may appear to be more convenient for users, however, it may require the user to have extensive knowledge of group membership. Additionally, it does not allow for very efficient access evaluation as discussed in the following section.

8.3 System-Defined Ordering

In this method, entries are considered according to a system-defined ordering. Although the user does not have the flexibility of choosing an arbitrary order of entries, a system-defined ordering gives consistency to ACLs throughout the system and may also allow for quicker access determination.

The system may use any of a variety of ordering methods, two of which are alphabetical ordering by user or group name and numeric ordering by user or group ID. An ordering of lowest to highest UID or GID, or vice-versa, is recommended as it provides an efficient way to check for redundant entries. Redundant entries should not be allowed in an ACL.

It is important to mention that actual sorting need not be done by the kernel itself as long as the kernel enforces the specified ordering. In other words, the sorting can be achieved through the use of library routines. The ACL commands would automatically use the library sorting routines and users would also be encouraged to

do so when writing their own programs which manipulate ACLs. When an ACL is passed to the kernel, the kernel verifies that the entries are sorted or else a failure will occur. In this manner, efficiency is achieved while still enforcing a system-defined ordering.

This alternative is simple, reduces the possibility of user error, and allows for more efficient access determination.

8.4 Multiple Group Evaluation

When a subject is a member of multiple groups, there are several ways the group entries may be evaluated, regardless of the ordering of the entries.

The following methods may be used to evaluate access when multiple groups are used:

The first entry which matches one of the subject's groups might be used to determine access. While this is an efficient method, it does not take notice of the possibility of other groups granting access.

The entry which matches one of the subject's groups and grants the least access might be used. This method does not recognize the possibility that all the groups together might grant or deny the desired access.

The entry which matches one of the subject's groups and grants the most access might be used. This method also does not recognize the possibility that all the groups together might grant or deny access.

ANDing the permissions of all the entries which match groups of the subject is another possible method. This approach may be considered too restrictive, since even one entry which grants access may be overruled by other entries which deny access.

ORing the permissions of all the entries which match groups of the subject is also a possibility. This method may be considered too permissive, since the maximum permissions allowed by all the matching entries taken together is the result. However, the same effect can be achieved currently, through the user simply invoking the *newgrp* command to change to the group with the desired access or by opening the same file twice from two different groups which together provide the desired access.

8.5 Recommendation

A system-defined ordering which evaluates ACLs entries from most-to-least specific is recommended. Since multiple groups were designed to be permissive and

permissive results can be achieved through other means anyway, the method which ORs the permissions of all matching group entries is recommended for systems implementing multiple groups.

Concern has been expressed that this scheme violates the wording in the TCSEC, for DAC at class B3. The TCSEC states: *Furthermore, for each such named object, it shall be possible to specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given.* The ORing of groups, however, does not present a conflict with the class B3 DAC requirement, as it still allows the user to specify groups that shall have no access.

9. DAC Compatibility

Designing an ACL mechanism requires that attention be given to the use of system calls which check or modify the existing DAC mechanisms, and to the additional use of ACL mechanisms in system calls. The classes of DAC mechanisms which return or change the value of the discretionary access control information are those mechanisms which: change ownership of an object, change the file permission bits, create objects, access object attributes, and access object data. Each of these classes will now be examined and a determination will be made of what changes, if any, are required for inclusion in a system with ACLs. For each class, we provide alternative solutions and identify the preferred choice.

9.1 Changing Ownership Of An Object

Mechanisms which change ownership of an object (e.g., **chown**, **msgctl**, **semctl**, **shmctl**) could create a new user or group entry for the object owner or group, with the same access permissions as the original entry for the object owner or group. The original entry would become an additional user or group entry. The problem with this alternative is that by leaving the original entry for the object owner or group behind as an additional user or group entry, the mechanism will always create an ACL for an object which did not have one to begin with.

The preferred alternative is for these calls to suffer no additional side effects due to the presence of ACLs. This can be achieved by not storing explicit IDs in the owner and owning group ACL entries. An advantage of this alternative is that the ACL entries for object owner and object owning group can be readily distinguished syntactically from the other user and group entries.

9.2 Changing The File Permission Bits

Mechanisms which change the file permission bits (e.g., **chmod**, **msgctl**, **semctl**, **shmctl**) might be changed so that they fail, or partially fail, when presented with an

object that has an ACL.

Complete failure is a poor alternative since these mechanisms change the file mode, not just the file permission bits. For example, a program should be able to do a legitimate operation such as changing the setgid bit on any file.

Partial failure means that these mechanisms would make the requested changes but return an error value different from -1. This is a poor alternative for two reasons: it does not make good sense to succeed while returning failure, and programs often do not differentiate between error return values.

Other alternatives attempt to minimize surprises to the caller by changing ACL entries. The first of these alternatives is to mask the access permissions in all the object's additional entries. Access permissions for entries with specific user and specific group are ANDed with the supplied user and group access permissions. Access permissions for entries with only a specific user are ANDed with supplied permissions for the user, and permissions for entries with only a specific group are ANDed with supplied permissions for the group. While this meets POSIX requirements, programs that wish to change only the file mode (non-access) bits will have the masking occur as an undesirable side effect. Another alternative is to disable the additional entries. This implicitly requires a new mechanism to enable entries that have been disabled. POSIX requirements are also satisfied by this alternative, but the same problems exist as in the previous alternative; programs using these mechanisms to change the non-access file mode bits will have entries disabled as an undesirable side effect. Still another alternative is to delete the additional entries. This has similar advantages and disadvantages as ACL entry disabling. It is simpler since there is no need for an ACL entry enabling mechanism. Information given by the user, however, is deleted without warning.

The preferred method is to make no changes to these mechanisms. The mechanisms will affect only file permission bits and ACL entries for the object owner or group. While this does not provide non-ACL cognizant programs with expected results for operations on objects with ACLs, it is not perceived as a serious problem. This alternative is consistent with the preferred alternative for mechanisms which access object attributes as well (see below).

9.3 Creating Objects

Mechanisms which create or truncate objects (e.g., **creat**, **open**, **mkfifo**, **mknod**, **msgget**, **semget**, **shmget**) should work as they currently do, except that they may create an ACL as part of the default ACL mechanism. Please refer to the section on default ACLs for more information. Note that default protection on newly-created objects will be accomplished via the umask and/or default ACLs.

It may also be desirable to add other types of ACL features to mechanisms. For example, one might wish to add the capability during file creation to adopt a specific ACL. For changes of this type, parameters of existing mechanisms should not be changed, and new parameters should not be added. New mechanisms should be created which make use of existing ones. For example, **creat** may need to be modified to take ACLs into account, but the parameter list should not change. Instead of adding an ACL parameter to **creat**, a new system call (i.e., with some other name) should be used, which takes the ACL as a parameter and then uses **creat**.

9.4 Accessing Object Attributes

Mechanisms which access object attributes (e.g., **stat**, **msgctl**, **semctl**, **shmctl**) could be modified to fail when applied to an object with an ACL. This is an unacceptable alternative since these mechanisms return more information than simply the file mode. Thus, non-functionality would require a new mechanism to return the additional information for objects with ACLs.

Another alternative is to find all the entries in the ACL that apply to the user-ID and group-ID of the subject, just like a permissive access check. Then OR all the associated permissions together, and return the results in the appropriate file permission bits (user, group, and other). While this alternative integrates the idea of ACLs into mechanisms that access object attributes, the context of the mechanisms affects the result returned to the point where the meaning of what the mechanisms return is somewhat clouded.

The preferred alternative is to make no changes to these mechanisms. The mechanisms will continue to return the file permission bits, as if ACLs did not exist. Another mechanism must then be used to find out if the file has an ACL, and if so, what its entries are. While this alternative does not provide all information to subjects that don't know about ACLs, it does not change the current behavior of these mechanisms.

9.5 Accessing Object Data

There are a number of system calls which will need to have ACL functionality added to them (i.e., for access checking). These calls include all those taking file system object names as parameters, as well as those IPC mechanisms which perform access checks. Examples of some of these calls are: **open**, **msgsnd**, **msgrcv**, **semop**, and **shmat**.

It is also important for portability that programs use the available access control mechanisms in an appropriate manner, so that the security policy is interpreted

correctly. For instance, at the system call level, the permission information returned by the use of **stat** may not be sufficient to determine allowed access; other information such as ACL contents may have to be evaluated as well.

9.6 Recommendation

The following is a summary of the preferred alternatives stated in this section. Regarding compatibility with existing DAC mechanisms that either 1) change ownership or group of an object, 2) change file permission bits, or 3) access object attributes should remain unchanged and not affect an existing ACL on the object or create an ACL where one did not exist before.

Regarding the addition of ACL functionality, existing mechanisms should not be changed, and new parameters should not be added. Instead, new mechanisms should be created which make use of existing ones.

10. ACL System Calls And Commands

This issue addresses what the naming conventions and functionality for ACL system calls and commands should be.

For system calls, there are at least two alternative types of designs. Each depends on how the ACL is viewed. In one approach, the ACL is a series of independent records which can be individually manipulated using calls similar to **open**, **read**, **write**, and **close**. This approach has a nice parallel to the way files are read and written, but may be viewed as overly complicated given the relative infrequency of ACL modification. In the other approach, the ACL is considered a single unit and is not changed record-by-record, but instead always manipulated as a whole. This approach uses a "get" and "set" concept for ACL operations, where an ACL, as a whole, is retrieved, modified locally, and then replaced [3]. This approach is simple and reflects the growing trend towards get/set type operations.

It may also be reasonable to extend the "get" and "set" concept to apply to default ACLs as well as to the ACL associated with an object. This is a natural extension of the way ACLs would be manipulated, and default ACL operations may be easily added to the recommended system call interface described below.

There are also two possible methods for implementing these calls. One option is to use separate system calls for each of the ACL operations (i.e., **getacl**, **setacl**). The other option is to have one ACL system call that can be invoked with a number of command arguments indicating the desired ACL operation [3]. An example of a useful additional command argument is one that would return the number of entries in the ACL. This method conserves the number of system calls, and provides the

flexibility to add ACL commands via command arguments. Additionally, using this method, designers are free to implement library functions based on the system call with particular command flags.

For commands, the same issues apply as for system calls. In a system with ACLs, however, there will be a need for commands to not only manipulate ACLs, but also to show and manipulate all discretionary access control information. These commands should include, at a minimum:

- command(s) to retrieve and set file permission and mode bits (**ls**, **chmod**)
- command(s) to retrieve and set ACL information (**new**)
- command(s) to retrieve effective discretionary access to files (**new**)

In addition, there may be useful features to add to existing utilities (e.g., the ability to find a file according to its ACL [12]) so that they might be able to conform to the enhanced DAC mechanisms.

10.1 Recommendation

For the ACL system call interface, get/set ACL type operations should be used, and should be implemented with a unified system call with command arguments used to implement the various operations. For commands, the names **getacl** and **setacl** are recommended since they follow from the get/set concept.

11. Named ACLs

A named ACL, as described in *A Guide to Understanding Discretionary Access Control in Trusted Systems* [2], is an ACL that can be shared or referred to by name. They may be implemented in one of two ways; either as a template copied into a user's ACL or shared through a pointer from the user's ACL space (*shared ACL*).

A change to a shared ACL results in a change to the discretionary access on all objects using this ACL. This result may be considered to be a side-effect or a desired feature depending on the circumstance. Additionally, it may be difficult to determine which objects are sharing a specific named ACL, and a user may mistakenly grant access to an object that was not intended.

Another problem with named ACLs is that as objects they may themselves be required to contain discretionary access controls. This suggests the idea of recursive ACLs, a situation to be avoided.

11.1 Recommendation

Named ACLs need not be supported, but a system that does should be no less secure or less flexible than one that does not. Absolute flexibility of ACLs can be achieved, however, through the use of default ACLs as discussed in the following section. There is no strong case one way or the other for named ACLs. There are advantages and disadvantages to both alternatives and it would really depend on the environment as to whether named ACLs would be of any benefit.

12. Default ACLs

When considering ACLs, an issue arises as to whether a predesignated set of ACL entries should be assigned to an object automatically at the time of creation. The following alternatives present the possible ways to address this issue.

12.1 No Default ACLs

In this approach, no ACL is assigned at object creation time. The process *umask* will limit the file permission bits, as it currently does, to provide some default protection on an object.

While this alternative maintains compatibility with existing programs, it is not a very practical solution. Depending on the relationship of the file permission bits and the ACL, the absence of default ACLs may not make sense. For instance, in a pure ACL implementation, the absence of default ACLs would result in no initial protection on newly created files. Additionally, this alternative would not encourage the use of ACLs by new programs, and would prevent ACL creation by old programs. ACLs could not propagate through the system and hence their usability would be lost.

12.2 Require Default ACLs

In this approach, an ACL would always be assigned at object creation time. This would allow for initial finer grained control on an object.

Requiring default ACLs may cause incompatibilities for an old program that only looks at the file permission bits when it creates an object. Also, for many users, the *umask* may be a sufficient tool for limiting the permissions on an object when it is created. The main advantage of requiring default ACLs is that the usability of ACLs is greatly improved. Additionally, since an ACL is associated with an object in a single atomic operation, the possibility of a temporarily insecure state is avoided.

12.3 Provide Default ACLs

A mechanism is provided to put default ACLs on new objects. However, not all new objects need to have default ACLs. This alternative allows specification of a default ACL, giving a finer granularity of access control than that provided by the file permission bits, and, at the same time allows, where desired, compatibility with existing programs.

12.4 Recommendation

Providing default ACLs and mechanisms to specify whether or not to use them is the best solution. This allows both classes of users, those who want default ACLs and those who do not (even those who want no ACLs at all), the flexibility to specify the scheme that they find most appropriate. Although in many cases the process umask would be sufficient to assign default permissions, systems and/or users making explicit use of ACLs will desire default ACLs. The default ACL scheme used should be straightforward to the user and should sensibly interact with the existing DAC mechanisms, including the *umask* mechanism. Note that even if an object is created with no default ACL, ACL entries may still be added to the object.

This section has really only addressed default ACLs on file system objects. IPC objects are not part of the file system name space, and therefore require further consideration. IPC objects are relatively short lived, and are generally not manipulated by users at the command level as are files. Based on these characteristics default ACLs on IPC objects are probably not needed, and their use is not recommended.

13. Location Of Default ACLs

Consider the following possibilities for the origination of the default ACL.

13.1 System Wide

In this approach, one specific default ACL is assigned to any object created on the system by any subject. This is a very inflexible solution and misses the intent that discretionary access be set at the discretion of the user.

13.2 Per Process

In this approach, each user process defines a default ACL, similar to the umask currently used. This is a somewhat restrictive approach since this allows the user to set only a single set of defaults for all files created. It is likely that a user will wish to associate different default ACLs with files created for different projects. Additionally, the default ACL entries would have to be stored in the process area.

The amount of process space required to hold the entries would vary based on the number of entries.

13.3 Per GID Of Created File

A default ACL could be associated with each GID. If GIDs are viewed as project identifiers, the effect is to associate a unique default ACL within each project subtree of the file system hierarchy. Further, in some UNIX Systems, where GIDs propagate to newly created objects based on the GID of the creating directory (rather than upon that of the creating subject), default protection very naturally distributes across the file system. However this variant imposes a somewhat restrictive viewpoint on the utility of groups.

13.4 Per Directory

This approach would allow the object's default ACL to originate from the containing directory of the object. A directory would contain both an ACL to be used for access checking and a default ACL to be used when a new object is created in the directory. All objects created in the directory would be assigned the default ACL. Newly created subdirectories would inherit the default ACL of the parent directory. In this manner, the default will propagate down through the file system structure resulting in much duplication of ACLs, possibly using much space. However, the utilization of such space is a small price to pay for enhanced security and usability, so the default should probably continue to propagate until the user takes some explicit action to stop the propagation.

13.5 Recommendation

A user typically arranges objects per directory representing project work or areas of interest. Since it is desirable, then, for similar objects to contain the same ACL, the per-directory approach becomes the preferred mechanism. Newly-created subdirectories should inherit the default ACL of the parent directory, so that defaults are propagated down the file system, unless explicitly turned off.

14. Interaction Of Default ACL Entries At File Creation

Currently, when a file is created a user can specify its initial permissions, however the access can be further restricted by the *umask* mechanism. The *umask* specifies the default protection bit settings when a file is created. Any bits set in the *umask* will be cleared in the bit settings on the newly created file. It is important, then, to consider how the default permission bit settings should interact with the entries in a default ACL.

Consider the following options in the context of masking the ACL entries by the file group class permission bits as recommended in the *ACL Evaluation* section. Also note that these options are discussed with respect to the ACL entry types as described in the *ACL Entry Type and Format* section. Additional mechanisms in the ACL which allow direct modification of the file group class permission bits at file creation are not precluded.

14.1 OR File Group Class Permission Bits

Add the default entries to the file and change the file group class permission bits to reflect the maximum permissions allowed in the ACL. This could result in more permission than was specified in the creation call. It is not reasonable to assume that the default permission bit settings can be ignored and completely overridden by the ACL. For example, if a default entry exists for user "fred" with the specified permissions of "rwx" but the file is not executable, then this permission should not be given.

14.2 AND File Group Class Permission Bits

Add the default entries to the file but change the permissions of the ACL entries so that they are no greater than the file group class permission bits. This is a reasonable alternative, but it may present a compatibility problem for some applications. An example of this problem would be when a C compiler creates a file. The file would not originally be created with execute permission, therefore no ACL entries on the file (which were default entries copied from the directory) would have execute permission. The last step for the compiler would be to make the file executable, however at this point, execute permission which may have been specified in the default ACL entry is lost.

14.3 No Change To File Group Class Permission Bits

Add the default entries to the file but do not change the file group class permission bits. This may result in ACL entries which are restricted by the file group class permission bits.

14.4 Recommendation

The *No Change To File Group Class Permission Bits* is recommended since it is a reasonable alternative which does not present any problems of compatibility for some applications.

15. Summary

This document has provided an analysis of key issues involved in extending the discretionary access control in the UNIX system. For each of the issues identified, the paper has suggested alternative solutions, discussed the pros and cons of each, and then provided a recommendation.

The following is a review of some of the important recommendations presented in the paper. An access control list mechanism was chosen to extend the current DAC mechanism. When considering the types of access provided in the UNIX system, additional access modes need not be defined, however they should also not be precluded. The recommended ACL entry type was that of user or group entries. The main advantages of this solution are conformance with the UNIX system method of identification through either the user-ID or the group-ID, and simplicity for the user. The method in which file protection bits and ACLs interact is a very important and complex issue given the conflicting goals of security and compatibility. The recommendation of masking the ACL entries by the group field of the protection bits was chosen as the most accommodating solution considering these goals. A system defined ordering of the ACL entries was preferred and it was recommended that the access allowed for a user in multiple groups should be the sum of all access allowed for each group represented in the ACL. Considering other multiple group issues, it was recommended to provide the multiple concurrent group capability along with some method of subsetting. It was also recommended that default ACLs be provided and that they originate from the parent directory of the newly created object.

It is important to note that although these and other specific recommendations were given, it is certainly possible to design an acceptable class B3, POSIX-compliant UNIX system following some of the other alternatives. In fact, there are issues where the recommended solution may not be superior to another alternative and the designer should consider his/her own specific requirements when making a choice in those areas. It must also be pointed out that building a system following all the recommendations presented in this paper will not guarantee a full class B3 system. There are many additional class B3 requirements that go beyond the interface specification.

APPENDIX: Worked Example

A.1 Introduction and Overview

This worked example describes one particular implementation following the recommendations in the TRUSIX rationale.

A.1.1 Discretionary Access Control

Discretionary access control (DAC) provides for the controlled sharing of objects (e.g., files, IPC objects) between subjects (e.g., processes). With discretionary access control, the owner of an object can grant permissions to other users. The discretionary access control mechanism uses object owner, object group, file permission bits (nine permission bits) and the access control list (ACL) of an object to determine the discretionary access to the object.

This document will detail the DAC interfaces and their run-time behavior.

The goals of this ACL mechanism were:

- compatibility with the current UNIX System DAC mechanism and POSIX P1003.1
- user command interfaces that are easy to use and understand
 - adhere to the "principle of least astonishment"
- interfaces should continue to work as expected
 - `chmod 000 file` - no access to file
 - `chmod 700 file` - only owner access to file
 - `chmod 444 file` - denies write and execute access to file

In addition, intermixing use of the existing and new DAC commands should give reasonable results. For instance `chmod` should not fail due to ACLs, and when `chmod x file` is executed (*x* is an octal permission) `ls -l` displays *x* as the permissions.

The current output of `ls -l` displays the file permission bits as a constant width set of nine characters:

```
rw-rw-rw-
```

However, an ACL, which consists of one or more **user** entries, one or more **group** entries, one **class** entry, and one **other** entry, is not a constant length (in the following example, * indicates zero or more occurrences of the preceding entry type):

```
# file: filename
# owner: uid
# group: gid
user::rwx
user:uid:rwx
*
group::rwx
group:gid:rwx
*
class:rwx
other:rwx
```

The file permission bits shown by the ls command have the following meaning: (note the following "class" definitions are from the IEEE POSIX Std 1003.1-1988):

1. the first 3 bits (high order) represent the file owner class and define the permissions for the object owner,
2. the middle 3 bits (commonly called the group permission bits), represent the file group class. This class includes the owning group of the file and will be extended to include additional **user** and additional **group** ACL entries,
3. the last 3 bits (low order) represent the file other class and define the permissions for other (those that did not fall into 1 or 2 above).

These nine bits indicate the maximum discretionary permissions for an object. The actual permissions may always be less than indicated. For instance, the permission may indicate write access on an object by a specific subject, but the file system may be mounted read only. If an ACL mechanism is used these bits will continue to indicate the maximum discretionary permissions for the object and the ACL may further restrict permissions.

There is a direct mapping between the ACL and the file permission bits. Specifically, the file owner class permission bits will always be equal to the permissions of the ACL entry for the object owner (they may be the same bits depending upon the implementation). Additionally, the file other class permission bits will always be equal to the ACL **other** entry permissions. And the file group class permission bits will always be equal to the ACL **class** entry permissions. Typically, the file group class permission bits are set to the maximum permissions allowed to the additional **user** entries, the owning **group** entry, and the additional **group** entries.

Whenever a file is created on a file system that supports ACLs, the ACL will contain a **user** entry for the object owner, a **group** entry for the object owning group, a **class** entry for the file group class permissions, and an **other** entry for the rest of the world. For compatibility with the current mechanism, if the ACL contains no additional **user** or additional **group** entries, the permissions in the **group** entry for the object owning group and the **class** entry must be the same.

A.1.2 Use of Access Control Lists

The use of DAC with ACLs will be explained by comparing it to how a user of a non-ACL supporting UNIX System (as currently exists) would use DAC. To use the current DAC mechanism a user usually first executes *ls -l* and based on the output decides what the permissions must be changed to, in order to allow the desired access (for example the user may want to make the file executable, or only allow the owner to have write permission).

EXAMPLE:

```
$ ls -l foo
-rw-rw-rw-  1 craig  demo    53 Mar  6 17:37 foo

$ chmod 600 foo

$ ls -l foo
-rw-----  1 craig  demo    53 Mar  6 17:37 foo
```

In the new DAC mechanism, using a pure ACL, there will be two new commands *getacl* and *setacl* (there will be a new function, *acl*, for which these commands provide a user interface). The *getacl* command will be used to display the ACL and the *setacl* command will be used to change the ACL.

These commands will be used in much the same way that *ls* and *chmod* are used. A user would first execute *getacl* to look at the ACL and then use *setacl* to make the desired changes. Because the ACL is not a fixed size, it may be difficult to manipulate. In order to simplify the use of ACLs the following example shows how the ACL may be easily manipulated using a text editor to give greater flexibility (note that changes may also be specified on the *setacl* command line).

EXAMPLE:

#the output of getacl is redirected to the file tmp

```
$ getacl bar > tmp
```

#the file tmp is edited and the line in *italics* is inserted

```
$ vi tmp
```

```
# file: bar
```

```
# owner: craig
```

```
# group: demo
```

```
user::rw-
```

```
group::rw-
```

```
group:guest:r--
```

```
class:rw-
```

```
other:rw-
```

#setacl is executed and the contents of the file tmp become the new ACL for bar

```
$ setacl -f tmp bar
```

#the output from getacl for the file bar is displayed

```
$ getacl bar
```

```
# file: bar
```

```
# owner: craig
```

```
# group: demo
```

```
user::rw-
```

```
group::rw-
```

```
group:guest:r--
```

```
class:rw-
```

```
other:rw-
```

A.1.3 Structure of Access Control Lists

The ACL consists of the following types of entries, which must be in the following order:

1. **user** entry - This type of entry contains a user ID and the permissions associated with it. There must always exist one entry of this type, which will represent the object owner, and will be denoted by a null (unspecified) user ID. There may be additional **user** entries specified; however, no two additional **user** entries will have the same user ID and there may not be any additional entries with a null user ID. The term "additional **user** entries" will be used to indicate all **user** entries except the entry for the object owner.

2. **group** entry - This type of entry contains a group ID and the permissions associated with it. There must always exist one entry of this type, which will represent the object owning group, and will be denoted by a null (unspecified) group ID. There may be additional **group** entries specified; however, no two additional **group** entries may have the same group ID and there may not be any additional entries with a null group ID. The term "additional **group** entries" will be used to indicate all **group** entries except the entry for the object owning group.
3. **class** entry - This type of entry contains the maximum permissions granted to the file group class. There is exactly one of these entries in an ACL.
4. **other** entry - This type of entry contains the permissions granted to a subject if none of the above entries have been matched. There is exactly one of these entries in an ACL.
5. **default** entry - This type of entry may only exist on a directory. These entries are similar to the entries described above, except that they are never used in an access check, but are used to indicate the non-default ACL entries that should be added to a file created within the directory. Default entries are optional, but no two default entries may have the same type and ID.

Within each category the entries must be ordered as follows:

Entries in the **user** category shall be sorted numerically by user ID from lowest to highest, except for the object owner entry, which always precedes all other user entries.

Entries in the **group** category shall be sorted numerically by group ID from lowest to highest, except for the object owning group entry, which always precedes all other group entries.

Entries in the **default:user** category shall be sorted numerically by user ID from lowest to highest, except for the default object owner entry, which always precedes all other default user entries. Entries in the **default:group** category shall be sorted numerically by group ID from lowest to highest, except for the default object owning group entry, which always precedes all other default group entries.

The proper ordering of entries required by the *acl* function can be obtained by the use of the *aclsort* function. ACL entries given as input to the *setacl* command need not be sorted; the sorting will be performed by the *setacl* command.

The permissions that may be specified in an ACL entry are read(r), write(w), and execute/search(x).

When the *setacl* command is executed, the file owner class permission bits will be set to the permissions specified for the owner and the file other class permission bits

will be set to the permissions specified for **other**. As an option, the file group class permission bits will be manipulated such that they reflect the maximum permission that the ACL permits to members of the file group class (any ACL entry other than the object owner or **other**). Otherwise, the file group class permission bits will be set to the permissions specified by the **class** entry. Therefore, if the file group class only allows read permission then additional **user** entries and any **group** entries in the ACL will not grant write or execute permission.

This ACL scheme supports finer discretionary access controls than the current mechanism, while maintaining compatibility with the current permissions mechanism. The DAC information may be changed in one atomic operation, avoiding the possibility of an intermediate insecure state. Finer controls can be specified via the ACL, including explicit specification of users disallowed any access to the object. Additionally, the file permission bits provide a summary of all access rights.

Rationale: The ACL scheme described here will allow entries to be either permissive or restrictive. In general, an entry that results in less permission than the file other class permissions would grant would be considered restrictive. An entry that results in more permission than the file other class permissions would grant would be considered permissive. In the event that a file with an ACL is exported to a non-ACL system, the loss of permissive entries would not present a security problem; however, the absence of support for restrictive entries may allow a process to have permission that it would not have been granted on a system with ACLs. This behavior must be described in the documentation.

A.1.4 Discretionary Access Check Algorithm

A process may request read, write, or execute/search access permissions to a file. Each access mode is logically checked separately using the following algorithm. The process request is granted if all individually requested modes are granted. Otherwise, the access request is denied.

Note, this is a logical description of the access check. The physical code sequence may be different for better performance.

Discretionary Access Check Algorithm:

- I. **File Owner Class:** If the effective user ID of the process matches the user ID of the owner of the file, the process is in the file owner class. If the requested access mode bit is set in the file owner class permission bits, this access mode is granted. Otherwise, access is denied.

Note, the **user** ACL entry for the object owner matches the file owner class permission bits.

- II. File Group Class: If the process is not in the file owner class and if the effective user ID of the process matches the user ID of an additional **user** ACL entry or the effective group ID or any of the supplementary group IDs of the process matches the group ID of any **group** ACL entry, the process is in the file group class. If the process matched an additional **user** ACL entry, only that entry is used as the matching ACL entry; otherwise, the matching **group** ACL entry or entries are used. If the requested access mode bit is set in the file group class permission bits and is set in a matching ACL entry, this access mode is granted. Otherwise, access is denied.

Note, the permissions of the additional **user** or **group** ACL entries further restrict the access specified by the file group class permission bits. Also, the **class** ACL entry matches the file group class permission bits.

- III. File Other Class: If the process is not in the file owner class or file group class, the process is in the file other class. If the requested access mode bit is set in the file other class permission bits, this access mode is granted. Otherwise, access is denied.

Note, the **other** ACL entry matches the file other class permission bits.

The following examples show ACL use and the results of applying current and new DAC commands.

EXAMPLE 1:

```
#create file foo
$ > foo
#execute ls -l and getacl on the file foo
$ ls -l foo
-rw-r--r--  1 craig  demo    0 Mar  6 20:27 foo

$ getacl foo
# file: foo
# owner: craig
# group: demo
user::rw-
group::r--
class:r--
other:r--
```

EXAMPLE 2:

#execute getacl and ls -l on the file, run.sh, with added ACL entries

```
$ ls -l run.sh
```

```
-rwxr-xr-x+ 1 craig  demo    73 Mar  6 20:27 run.sh
```

```
$ getacl run.sh
```

```
# file: run.sh
```

```
# owner: craig
```

```
# group: demo
```

```
user::rwx
```

```
user:fred:r-x
```

```
user:larry:--x
```

```
group::r-x
```

```
group:guest:---
```

```
class:r-x
```

```
other:r-x
```

EXAMPLE 3:

#use the chmod command on a file with an ACL

#use getacl to report both the ACL entries and the effective permissions

```
$ chmod 644 run.sh
```

```
$ ls -l run.sh
```

```
-rw-r--r--+ 1 craig  demo    73 Mar  6 20:27 run.sh
```

```
$ getacl run.sh
```

```
# file: run.sh
```

```
# owner: craig
```

```
# group: demo
```

```
user::rw-
```

```
user:fred:r-x      #effective:r--
```

```
user:larry:--x     #effective:---
```

```
group::r-x         #effective:r--
```

```
group:guest:---
```

```
class:r--
```

```
other:r--
```

A.1.5 File Object Creation

When a new object (regular files, special files, directories, named pipes) is created in the file system, there are several important attributes that must be initialized. These

are the user ID of the owner of the file, the group ID associated with the file, the file permission bits, and the ACL.

The user ID of the file is set to the effective user ID of the invoking process. The group ID of the file depends upon the mode of the containing directory. If the S_ISGID bit is not set on the directory, the group ID of the file is set to the effective group ID of the invoking process. If the S_ISGID bit is set on the directory, the group ID of the file is set to the group ID of the containing directory.

Each function that creates a new file supplies an initial value for the file permission bits. This initial value is then merged with the file mode creation mask (**umask**) of the invoking process and with any default ACL entries of the containing directory to form the file permission bits and ACL of the new file.

Although in many cases the process **umask** is sufficient to assign default permissions, users making explicit use of ACLs may desire default ACLs. The default ACL scheme must sensibly interact with the existing DAC mechanism, including **umask**.

The default ACL entries specify permissions for users and/or groups and/or others, that will be assigned to a new file. These default ACL entries are associated with a directory. Note, an ACL on a directory may contain entries that control access to the directory and entries (defaults) used for new file creation in that directory.

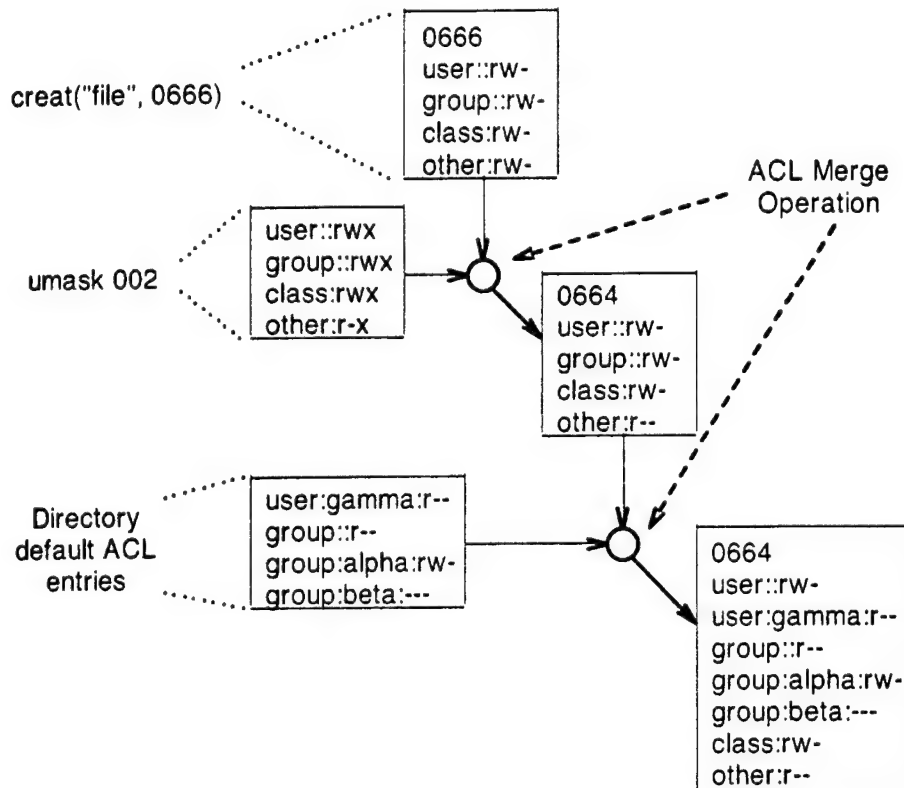
The process of creating the file permission bits and the ACL for the new file is called "ACL Merge". First, any mode parameter is transformed into the equivalent ACL form. For example, the mode 0664 is equivalent to user::rw-, group::rw-, class::rw-, other:r--. Also, the complement of the umask is used to obtain the equivalent ACL. Thus, the umask 022 is equivalent to user::rwx, group::r-x, class:r-x, other:r-x.

Two ACLs are merged by first logically sorting both ACLs into one ACL. Then any pair of matching entries are replaced with an entry that has permissions formed by ANDing the matched entries. Thus a permission is in the merged entry only if it was previously in both entries.

The first ACL merge is with the initial mode from the file creation function and with the process file mode creation mask. The second ACL merge is with any default entries from the containing directory. The result is the ACL for the new file. The file permission bits are then set from the **user**, **class**, and **other** ACL entries. Note, this may be different from the *setacl* command with the -r option since this merge does not set the file group class permission bits to the maximum permission of the file group class entries.

Finally, if the new object is a directory, then any default entries from the containing directory are copied to the new ACL. That is, the default ACL entries of the new directory are the same as the default ACL entries of the containing directory.

An example of the ACL merge operation is shown in the following figure:



A.1.6 IPC Object Creation

When an IPC object is created (by *shmget* for shared memory, by *semget* for semaphores, by *msgget* for messages), its cuid and uid will be set equal to the effective user ID of the invoking process and its cgid and gid will be set equal to the effective group ID of the invoking process. The initial permissions are set equal to the specified permissions in the flag argument to the **get* calls (*shmflg*, *semflg*, and *msgflg*, respectively). Note that default ACLs do not apply to IPC objects, although ACLs may be added explicitly to an IPC object via the **aclipc** call.

A.1.7 Compatibility Requirements

A user will generally use the current DAC commands (*ls* and *chmod*) or the new DAC commands (*getacl* and *setacl*). However, the use of these commands are likely to still be inter-mixed, and they must all give correct information.

The entire interface to the current discretionary access control information must

continue to function as it currently does. For example, *chmod* must still be able to modify the file permission bits and *ls* must still be able to report them.

Note that although *ls* will still report these permissions, they will not be the only permissions evaluated during an access check. The output of *ls* will continue to be the maximum permission that may be granted, but there may be additional discretionary access control information (ACL entries) that was added to the object. In order to indicate that additional entries exist, *ls -l* will display the character "+" to the right of the current permissions display if an ACL is present. Therefore, when additional discretionary access control information has been added, in the form of ACL entries (as shown in the examples on previous pages), a user will need to use the newly provided command, *getacl*, to get a full view of the current discretionary access controls in effect. Although *chmod* will still modify the file permission bits, it will not change any additional discretionary access control information (i.e., ACL entries for additional users and additional groups) added to the object. To change these additional entries if they exist, the user will need to use the *setacl* command.

When the owner of an object is changed, the result will be identical to the current behavior. If the owner is changed to a user ID for which an additional **user** entry already exists in the ACL, the additional **user** entry is not changed but the **user** entry for the object owner will take precedence during an access check. When the group of an object is changed, the result will be identical to the current behavior. If the group is changed to a group ID for which an additional **group** entry already exists in the ACL, the additional **group** entry is not changed but the **group** entry for the object owning group will take precedence during an access check (except in the case of multiple concurrent groups, where all group entries are given equal treatment).

When the ACL contains no additional **user** or additional **group** entries, the permissions in the **group** entry for the object owning group and in the **class** entry must be the same. This behavior is the same as the current mechanism since the file permission bits can only specify at most three different permissions.

A.1.8 Documentation Requirements

The ACL mechanism and its proper use must be fully described in the Trusted Facility Manual and manual pages must be created for the Security Features User's Guide and Security Features Programmer's Guide for all new commands and functions.

A.2 Commands and Functions

A.2.1 *setacl* Command

DESCRIPTION: The *setacl* command will support the changing of discretionary permission information associated with a file. It will allow the file owner or a process with appropriate permission or appropriate privilege to perform the following functions:

1. replace an entire ACL, including the default ACL entries on a directory,
2. add, change, or delete an ACL or default ACL entry or entries.

This command gives the user an interface to a pure ACL mechanism, allowing a finer granularity for file access.

Note that this command only supports the file system objects: e.g., regular files, special files, directories, and named pipes. For simplicity, these objects are referred to as "files".

SYNOPSIS:

```

setacl [-r] [ -m [u[ser]:operm | perm[,]]
                [u[ser]:uid:operm | perm[,...]]
                [g[roup]:operm | perm[,]]
                [g[roup]:gid:operm | perm[,...]]
                [c[lass]:operm | perm[,]]
                [o[ther]:operm | perm[,]]
                [d[efault]:u[ser]:operm | perm]
                [d[efault]:u[ser]:uid:operm | perm[,...]]
                [d[efault]:g[roup]:operm | perm]
                [d[efault]:g[roup]:gid:operm | perm[,...]]
                [d[efault]:c[lass]:operm | perm]
                [d[efault]:o[ther]:operm | perm]
        ]

[ -d [u[ser]:uid[,...]][g[roup]:gid[,...]] [d[efault]:u[ser]:[,...]]
    [d[efault]:u[ser]:uid[,...]] [d[efault]:g[roup]:[,...]]
    [d[efault]:g[roup]:gid[,...]] [d[efault]:c[lass]:[,...]]
    [d[efault]:o[ther]:[,...]]

```

file ...

or

```

setacl [-r] -s u[ser]:operm | perm[,]
        [u[ser]:uid:operm | perm[,...]]
        g[roup]:operm | perm[,]
        [g[roup]:gid:operm | perm[,...]]
        c[lass]:operm | perm[,]
        o[ther]:operm | perm[,]
        [d[efault]:u[ser]:operm | perm]
        [d[efault]:u[ser]:uid:operm | perm[,...]]
        [d[efault]:g[roup]:operm | perm]
        [d[efault]:g[roup]:gid:operm | perm[,...]]
        [d[efault]:c[lass]:operm | perm]
        [d[efault]:o[ther]:operm | perm]

```

file ...

or

```

setacl [-r] -f acl_file file ...

```


where:

operm = octal representation of permissions

(Note: for an ACL entry one octal digit is required)

perm = a permissions string composed of the

characters **r** (read), **w** (write), **x** (execute/search),

or **-** (no permission). The permission string must be at least 1 character and no more than 3 characters.

The characters **r**, **w**, and **x** may only be in the string at most once. The characters may be in any order within the string.

uid = user identity (i.e., login name or user ID)

gid = group identity (i.e., group name or group ID)

When the **-f** option is specified, it will take the access control information stored in the file *acl_file* and assign it to the file *file*. See the PROCESSING section below for further information on the format of the file *acl_file*.

PROCESSING: A unique ACL will exist for each file on the system. There are four types of ACL entries, consisting of **user**, **group**, **class**, and **other**. The **user** entry for the file owner, the **group** entry for the file owning group, the **class** entry for the file group class, and the entry for **other** must always be in the ACL.

1. **user** entry - This type of entry contains a user ID and the associated permissions that will be granted to the user. There must always exist one entry of this type, which will represent the file owner, and will be denoted by a null (unspecified) user ID. There may be additional **user** entries specified; however each entry must specify a unique user ID and there may not be any additional entries with a null user ID. If there is a **user** entry with a user ID equal to the file owner the file owner entry will take precedence when an access check is performed.
2. **group** entry - This type of entry contains a group ID and the associated permissions that will be granted to the group. There must always exist one entry of this type, which will represent the file owning group, and will be denoted by a null (unspecified) group ID. There may be additional **group** entries specified; however, each entry must have a unique group ID and there may not be any additional entries with a null group ID.
3. **class** entry - This type of entry contains the maximum permissions for the file group class. There is exactly one of these entries in an ACL.

4. **other** entry - This type of entry contains the permissions granted to a subject if none of the above entries have been matched. There is exactly one of these entries in an ACL.

When the *setacl* command is used to change the ACL, it may result in changes to the file permission bits. Specifically, when the **user** ACL entry for the file owner is modified the file owner class permission bits will be modified. When the **class** ACL entry is modified, the file group class permission bits will be modified. When the **other** ACL entry is modified the file other class permission bits will be modified.

When the additional **user** entries or additional **group** entries of the ACL are modified, the file group class permission bits may also need to be modified to reflect the maximum permission allowed by these entries.

The **-r**, recalculate, option will result in the permissions specified in the **class** entry being ignored and replaced by the maximum permission needed for the file group class. For example, if there are no additional **user** entries or additional **group** entries, the permission of the **group** entry for the file owning group is used for the **class** entry.

A directory may contain default ACL entries. These entries may be of the type **default:user**, **default:group**, **default:class**, or **default:other**. For **default:user** entries, if no user ID is specified, this entry will apply to the file owner permissions. Additional **default:user** entries must have a unique user ID specified. For **default:group** entries, if no group ID is specified, this entry will apply to the file owning group permissions. Additional **default:group** entries must have a unique group ID specified. If there are no additional **default:user** entries or additional **default:group** entries, then the permissions of the **default:group** and the **default:class** must be the same.

If a file is created in a directory which contains default ACL entries the entries will be added to the newly created file. Note that the default permissions specified for the file owner class, file group class, and file other class will be constrained by the **umask** and the mode specified in the file creation call. If default ACL entries are specified for a file which is not a directory the command will fail {11}, see **ERRORS AND RETURNS**.

With no options and arguments {1}, see **ERRORS AND RETURNS**. If the MAC or DAC check fails when a request is made to modify the ACL {2}, see **ERRORS AND RETURNS**. If the file named *file* does not exist {6}, see **ERRORS AND RETURNS**.

If options are specified, the validity of the option-arguments will be checked. If an invalid option is specified {3a}, see **ERRORS AND RETURNS**. The arguments must be processed in the order specified (e.g., if the modify option is specified with a user, followed by the delete option with the same user, the entry will be deleted).

For the **-m**, **-s**, and **-d** options, if *uid* is not a valid login name or a valid user ID {3b}, or if *gid* is not a valid group name or a valid group ID {3c}, or if a specified *perm* is not **r**, **w**, **x**, **-**, or a specified *operm* is not an octal digit {3d}, see **ERRORS AND RETURNS**.

The **-m** option is used to add a new ACL entry or change an existing ACL entry.

If an entry already exists for the specified *uid* or *gid*, the specified permissions (*perm|operm*) will replace the current permissions. If an entry does not exist for the specified *uid* or *gid*, an entry will be created. Note that an entry with no permissions will result in the specified *uid* or *gid* being denied access (any permissions) to the file. To specify no access in an entry being modified or added, either 0 should be specified for *operm* or **-** should be specified for *perm*.

The **-s** option is used to replace the ACL information on a file. The effect of using this option is that all entries are removed, and replaced by the newly specified ACL. If **-s** is specified with **-d**, **-f**, or **-m** {5}, see **ERRORS AND RETURNS**. There must be exactly one **user** entry specified for the file owner, exactly one **group** entry specified for the file owning group, exactly one **class** entry specified for the file group class, and exactly one **other** entry specified. If there is no **user** entry specified for the file owner, or no **group** entry specified for the file owning group, or no **class** entry specified for the file group class, or no **other** entry specified {8}, see **ERRORS AND RETURNS**. There may be additional **user** ACL entries and additional **group** ACL entries specified. If duplicate entries are specified {9}, see **ERRORS AND RETURNS**.

The **-d** option is used to delete an existing entry from the ACL. If a matching entry is not found {4a}, see **ERRORS AND RETURNS**. Otherwise, the matching entry will be deleted. The **user** entry for the file owner, the **group** entry for the file owning group, the **class** entry, and the **other** entry may not be deleted from the ACL. If an attempt is made to delete one of these entries {4b}, see **ERRORS AND RETURNS**.

(Note: deleting an entry may have different effects than removing all the specified permissions for an entry. If an entry is deleted and a search is later done for the user or group identity that appeared in the entry, this identity

might match another entry and then be given the permissions specified in this other entry. If the original entry remained with no permissions and a search was done for this identity, the search might match this entry and the subject would be denied access.)

The **-f** option is used to assign the ACL information contained in the file named *acl_file* to the specified file(s). If **-f** is specified with **-d**, **-s**, or **-m {5}**, see **ERRORS AND RETURNS**. If the file named *acl_file* does not exist {6}, see **ERRORS AND RETURNS**. The file named *acl_file* must be readable by the invoking subject. If it is not readable {2}, see **ERRORS AND RETURNS**.

If the entire file named *acl_file* contains correct external representation(s) for ACL entries, the ACL for the specified file(s) will be (removed and) replaced with the ACL whose external representation is contained in the file named *acl_file*. Each external representation of an ACL entry, contained in the file named *acl_file*, must be on a separate line and must be in the following format:

```
u[ser]::operm | perm
[u[ser]:uid:operm | perm]
g[roup]::operm | perm
[g[roup]:gid:operm | perm]
c[lass]:operm | perm
o[ther]:operm | perm
[d[efault]:u[ser]::operm | perm]
[d[efault]:u[ser]:uid:operm | perm[,...]]
[d[efault]:g[roup]::operm | perm]
[d[efault]:g[roup]:gid:operm | perm[,...]]
[d[efault]:c[lass]:operm | perm]
[d[efault]:o[ther]:operm | perm]
```

The entries are not required to be in any specific order within the file.

There must be exactly one **user** entry specified for the file owner, exactly one **group** entry specified for the file owning group, exactly one **class** entry specified for the file group class, and exactly one **other** entry specified. If not, see **ERRORS AND RETURNS**. There may be additional **user** ACL entries and additional **group** ACL entries specified. If duplicate entries are specified {9}, see **ERRORS AND RETURNS**.

Validity checks are performed on all entries. If an invalid entry is encountered {7}, see **ERRORS AND RETURNS**. If the exact problem can be determined an additional message may be displayed {3b}{3c}{3d}, see **ERRORS**

AND RETURNS.

The character "#" will be used to indicate a comment. All characters starting with the #, to the end of the line will be ignored. Note that this includes any effective permissions (#effective:rwX) displayed by **getacl**.

This command may be executed on a file system that does not support ACLs. If ACL entries are specified which do not map into the base permissions {10}, see **ERRORS AND RETURNS**, otherwise the base permissions will be set.

ERRORS AND RETURNS: Following is a list of error conditions and the corresponding error message that should be output when this condition occurs.

```
usage: setacl [-r] [ -m [u[ser]:operm | perm[.]]
                [u[ser]:uid:operm | perm[,...]]
                [g[roup]:operm | perm[.]]
                [g[roup]:gid:operm | perm[,...]]
                [c[lass]:operm | perm[.]]
                [o[ther]:operm | perm[.]]
                [d[efault]:u[ser]:operm | perm]
                [d[efault]:u[ser]:uid:operm | perm[,...]]
                [d[efault]:g[roup]:operm | perm]
                [d[efault]:g[roup]:gid:operm | perm[,...]]
                [d[efault]:c[lass]:operm | perm]
                [d[efault]:o[ther]:operm | perm]
        ]
        [ -d [u[ser]:uid[,...]][g[roup]:gid[,...]] [d[efault]:u[ser]:]
          [d[efault]:u[ser]:uid[,...]] [d[efault]:g[roup]:[,...]]
          [d[efault]:g[roup]:gid] [d[efault]:o[ther]:]]
        file ...
```

or

```

setacl [-r] -s u[ser]:operm | perm[,]  

      [u[ser]:uid:operm | perm[,...]]  

      g[roup]:operm | perm[.]  

      [g[roup]:gid:operm | perm[,...]]  

      c[lass]:operm | perm[.]  

      o[ther]:operm | perm[,]  

      [d[efault]:u[ser]:operm | perm]  

      [d[efault]:u[ser]:uid:operm | perm[,...]]  

      [d[efault]:g[roup]:operm | perm]  

      [d[efault]:g[roup]:gid:operm | perm[,...]]  

      [d[efault]:c[lass]:operm | perm]  

      [d[efault]:o[ther]:operm | perm]

```

file ...

or

```
setacl [-r] -f acl_file file ...
```

{1} No options or arguments:

UX:setacl: ERROR: incorrect usage
usage: ...

{2} If MAC or DAC check fails on the specified file:

UX:setacl: ERROR: permission denied for "*file_name*"

{3} invalid option-arguments:

{a} incorrect/unknown option specified:

UX:setacl: ERROR: illegal option -- "*-option*"
usage: ...

{b} invalid user ID:

UX:setacl: ERROR: unknown user-id "*uid*"

{c} invalid group ID:

UX:setacl: ERROR: unknown group-id "*gid*"

{d} invalid permission:

UX:setacl: ERROR: unknown permission "*permission*"
usage: ...

- {4} invalid attempt to delete an ACL entry:
 - {a} attempt to delete a non-existent entry from an ACL:
 UX:setacl: ERROR: matching entry not found in ACL
 - {b} attempt to delete file owner, file owning group, **class**, or **other** ACL entries:
 UX:setacl: ERROR: file owner, file group, "class", and "other" entries may not be deleted
- {5} the options specified are mutually exclusive:
 UX:setacl: ERROR: incompatible options specified
 usage: ...
- {6} *file_name* does not exist:
 UX:setacl: ERROR: file "*file_name*" not found
- {7} an invalid ACL entry encountered in the file *acl_file*:
 UX:setacl: ERROR: "*acl_file*", line *line*; invalid ACL entry
- {8} required entry for file owner, file owning group, **class**, or **other** missing:
 UX:setacl: ERROR: required entry for file owner, file group, "class", or "other" not specified
 usage: ...
- {9} duplicate ACL entries specified:
 UX:setacl: ERROR: duplicate entries: "*acl_entry*"
- {10} the file system does not have ACLs, and additional entries are specified:
 UX:setacl: ERROR: only file owner, file group, "class" or "other" entries may be specified
- {11} the specified file is not a directory, and default entries have been specified:
 UX:setacl: ERROR: default ACL entries may only be set on directories

OUTPUT: None

A.2.2 *getacl* Command

DESCRIPTION: The *getacl* command will support the displaying of discretionary information associated with a file. It will allow the file owner or a process with appropriate permission or appropriate privilege to perform the following functions:

1. display the owner, group, and ACL for the specified file(s),
2. display the default ACL for a directory.

Note that this command only supports the file system objects: e.g., regular files, special files, directories, and named pipes. For simplicity, these objects are referred to as "files".

SYNOPSIS:

getacl [-ad] *file* ...

PROCESSING: With no arguments {1}, see **ERRORS AND RETURNS**. If MAC or DAC check fails when a request is made to display the ACL information {2}, see **ERRORS AND RETURNS**. With invalid options {3}, see **ERRORS AND RETURNS**. If the file named *file* does not exist {4}, see **ERRORS AND RETURNS**.

With the **-a** option specified, the filename, owner, group, and the ACL of the file will be displayed. With the **-d** option specified, the filename, owner, group, and the default ACL of the file will be displayed, if it exists. If the specified file does not support default ACLs (e.g., it is not a directory) only the filename, owner, and group will be displayed. With no option specified, both the ACL and the default ACL (if it exists) of the file will be displayed.

This command may be executed on a file system that does not support ACLs. It will report the ACL based on the base permission bits.

ERRORS AND RETURNS: Following is a list of error conditions and the corresponding error message that should be output when this condition occurs.

usage: *getacl* [-ad] *file* ...

{1} No arguments:

UX: *getacl*: ERROR: incorrect usage
usage: ...

{2} If MAC or DAC check fails when a request is made to display the ACL information:

UX:getacl: ERROR: permission denied for "*file*"

{3} incorrect/unknown option specified:

UX:getacl: ERROR: illegal option -- "*-option*"
usage: ...

{4} *file* does not exist:

UX:setacl: ERROR: file "*file*" not found

OUTPUT: When an ACL is displayed, the external representation of the ACL will be as follows:

```
# file: filename
# owner: uid
# group: gid
user::perm
user:uid:perm
group::perm
group:gid:perm
class:perm
other:perm
default:user::perm
default:user:uid:perm
default:group::perm
default:group:gid:perm
default:class:perm
default:other:perm
```

The ACL entries will be displayed in the order listed above (the **user** entry for the file owner, followed by zero or more additional **user** entries, followed by the **group** entry for the file owning group, followed by zero or more additional **group** entries, followed by the **class** entry for the file group class, followed by the entry for **other**). When the specified file is a directory the entries described above may be followed by default entries (the **default:user** entry for the file owner, followed by zero or more additional **default:user** entries, followed by the **default:group** entry for the file owning group, followed by zero or more additional **default:group** entries, followed by the **default:class** entry for the file group class, followed by the entry for **default:other**). Note that these default ACL entries are never used in an access check.

If more than one file is specified, a blank line will be displayed before the ACL of the next file is displayed.

The first line displays the name of the file, next the file owner, and then the file owning group. The **user** entry without a user ID indicates the permissions that will be granted to the owner of the file. The additional **user** entries indicate the permissions that will be granted to the specified user. The **group** entry without a group indicates the permissions that will be granted to the group of the file. The additional **group** entries indicate the permissions that will be granted to the specified group. The **class** entry indicates the permissions that will be granted to the file group class. The **other** entry indicates the permissions that will be granted to others.

The default entries (**default:user**, **default:group**, **default:class**, and **default:other**) may only exist for directories, and indicate the default **user**, **group**, **class**, and **other** entries respectively that will be merged with the ACL for a new file created within the directory.

The *uid* is a login name, or a user ID (only if there is no login name associated with the user ID); *gid* is a group name, or a group ID (only if there is no group name associated with the group ID); and *perm* is a three character string composed of the letters representing the separate discretionary access controls, **r** (read), **w** (write), **x** (execute/search), or the character **-**. The *perm* will be displayed in the following order: **rwX**. If a permission is not granted by this ACL entry, the placeholder, **"-"**, will appear. For example, if the user does not have write permission, but does have read and execute permission, **r-x** will be output.

The file group class permission bits constrain the ACL (represent the most access that any entry in the ACL may have). If a user executes the *chmod* command and changes the file group class permission bits this may change the permissions that would be granted based on the ACL alone. This behavior is necessary for the save-restore model (all permissions are temporarily removed via **chmod 000 file** and then restored) to work correctly. In order to indicate that the file permission bits are more restrictive than an ACL entry, *getacl* will display the ACL entry as described above with an additional tab followed by a sharp sign and the effective permissions.

Note that output from *getacl* will be in the correct format for input to *setacl*. Therefore, if the output is redirected into a file (e.g., *getacl junk > entries*), this file can be used as input to *setacl* (e.g., *setacl -f entries junk.new*). In this way, a user can easily assign one file's ACL information to another file.

EXAMPLES:

1) File with several ACL entries:

```
# file: fred
# owner: craig
# group: demo
user::rwx
user:spy:---
user:larry:rw-
group::r--
class:rw-
other:---
```

2) Same file, after a "chmod 700 fred":

```
# file: fred
# owner: craig
# group: demo
user::rwx
user:spy:---
user:larry:rw-      #effective:---
group::r--          #effective:---
class:---
other:---
```

3) Directory with ACL entries including default ACL entries:

```
# file: foodir
# owner: craig
# group: demo
user::rwx
user:spy:---
user:larry:rwx
group::r-x
class:rwx
other:r--
default:user::rwx
default:user:larry:rwx
default:user:worm:---
default:group:demo:r--
default:other:---
```

A.2.3 *acl* Function

DESCRIPTION: The *acl* call will support the getting and setting of discretionary permission information associated with a file. It will allow the file owner or a process with appropriate permission or appropriate privilege to perform the following functions:

1. get or set a file's ACL information in an atomic operation.
2. return the number of entries contained in a file's ACL.

Note that this call only supports the file system objects: e.g., regular files, special files, directories, and named pipes. For simplicity, these objects are referred to as "files".

SYNOPSIS:

```
#include <tbl.h>
```

```
int acl(const char *path, int cmd, int nentries, struct acl *aclbufp)
```

Three values for *cmd* will be supported: **ACL_SET**, **ACL_GET**, and **ACL_CNT**. The value of *nentries* is the number of ACL entries that can fit in the user-supplied ACL buffer for an **ACL_GET** or the number actually present for an **ACL_SET**; and *aclbufp* is a pointer to the user-supplied buffer of ACL entry structures. The buffer will consist of an array of four (**USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, and **OTHER_OBJ** entries are required) or more occurrences of the following structure:

```
struct acl {  
    int a_type;  
    uid_t a_id;  
    ushort a_perm;  
};
```

Twelve values of *a_type* will be supported to specify the type of entry: (six for access checking and six for defaults), **USER_OBJ**, **USER**, **GROUP_OBJ**, **GROUP**, **CLASS_OBJ**, **OTHER_OBJ**, **DEF_USER_OBJ**, **DEF_USER**, **DEF_GROUP_OBJ**, **DEF_GROUP**, **DEF_CLASS_OBJ**, and **DEF_OTHER_OBJ**.

When *a_type* is **USER** or **DEF_USER**, *a_id* will be a user id, and when *a_type* is **GROUP** or **DEF_GROUP**, *a_id* will be a group id. When *a_type* is **USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, **OTHER_OBJ**, **DEF_USER_OBJ**, **DEF_GROUP_OBJ**, **DEF_CLASS_OBJ**, or **DEF_OTHER_OBJ**, *a_id* will not be used. The permissions for the entry will be contained in *a_perm*.

PROCESSING: When the specified *cmd* is **ACL_CNT**, the return value from the call will be the number of ACL entries for the filename pointed to by *path*. The values of *nentries* and *aclbufp* will be ignored. If the user does not pass the DAC and MAC checks to see the ACL, the *acl* call will fail (see **ERRORS AND RETURNS**).

When the specified *cmd* is **ACL_GET**, the ACL information for the filename pointed to by *path* will be retrieved and the ACL entries will be placed in the buffer pointed to by *aclbufp*. The value of *nentries* is the number of entries that can be held in the allocated buffer. If the number of ACL entries in the ACL is greater than the value of *nentries* (that is, the buffer space allocated to hold the file's ACL entries is less than *nentries* times the size of an entry), the *acl* call will fail (see **ERRORS AND RETURNS**). On success, the return value from this call will be the number of ACL entries retrieved. On any error, the contents of the *acl* structures pointed to by *aclbufp* are indeterminate. If the user does not pass the DAC and MAC checks to see the ACL, the *acl* call will fail (see **ERRORS AND RETURNS**).

When the specified *cmd* is **ACL_SET**, ACL entries currently in the buffer pointed to by *aclbufp*, for the filename pointed to by *path*, will be set if all required checks are passed. The contents of *nentries* shall be the number of ACL entries in the buffer, pointed to by *aclbufp*, to be copied. On success, the return value from this call will be 0. If the invoking user does not pass the DAC and MAC checks to set an ACL, the *acl* call will fail (see **ERRORS AND RETURNS**). If an error occurs, either due to DAC and MAC checks or the validation check listed below, there will be no change to the current ACL information. Before the ACL entries are actually set, validation checks will be performed to determine that the ACL entries are in the following order:

- a) a user entry for the file owner (**USER_OBJ**),
- b) additional user entries (**USER**),
- c) a group entry for the file owning group (**GROUP_OBJ**),
- d) additional group entries (**GROUP**),
- e) a class entry for the file group class (**CLASS_OBJ**),
- f) an entry for other (**OTHER_OBJ**),
- g) default user entry for the file owner (**DEF_USER_OBJ**),
- h) default additional user entries (**DEF_USER**),

- i) default group entry for the file owning group (**DEF_GROUP_OBJ**),
- j) default additional group entries (**DEF_GROUP**),
- k) default class entry for file group class (**DEF_CLASS_OBJ**),
- l) default entry for other (**DEF_OTHER_OBJ**),

The entry in classes a), c), e), and f) must always exist. The entry for classes a), c), e), f), g), i), k), and l) do not use the *a_id* field. Classes b) and h) may contain zero or more entries and the entries must be sorted by *uid* (lowest to highest). Classes d) and j) may contain zero or more entries and the entries must be sorted by *gid* (lowest to highest). (this ordering should be done with the *aclsort* function).

Class g), h), i), j), k), and l) entries are only applicable for directories. If an attempt is made to set default ACL entries on a file that is not a directory, the call will fail (see **ERRORS AND RETURNS**).

Validation of the ACL will be performed. If entries containing duplicate *uids* or *gids* are found, or there is not exactly one **user** entry specified for the file owner, one **group** entry specified for the file owning group, one **class** entry specified for the file group class, and one **other** entry specified, or there are no additional **user** and **group** entries and the permissions of the **class** entry are not equal to the permissions of the **group** entry, or there are no additional **default:user** and **default:group** entries and the permissions of the **default:class** entry is not equal to the permissions of the **default:group** entry, the call will fail (see **ERRORS AND RETURNS**).

The file owner class permission bits will be changed, such that they are equal to the permissions specified for the **user** entry of the file owner in the ACL. The file group class permission bits will be changed, such that they are equal to the permissions specified for the **class** ACL entry. The file other class permission bits will be changed, such that they are equal to the permissions specified for the **other** ACL entry.

This function may be executed on a file system that does not support ACLs. With **ACL_GET** as the *cmd* it will report the ACL based on the file permission bits. With **ACL_SET** as the *cmd*, if ACL entries are specified which do not map into the file permission bits, see **ERRORS AND RETURNS**, otherwise the file permission bits will be set.

A design may constrain the maximum number of ACL entries that are written, with a system-wide tunable parameter, **aclmax**. If the number of

ACL entries exceeds the value of **aclmax** the function will fail (see **ERRORS AND RETURNS**).

ERRORS AND RETURNS: If the *acl* call is unsuccessful, a value of -1 will be returned and *errno* will be set to indicate the error. Only implementation-independent *errno*s are presented.

Under the following conditions, the function *acl* will fail and will set *errno* to the specified value (note: unless otherwise stated, the *errno* applies to **ACL_CNT**, **ACL_GET**, and **ACL_SET**):

ENOTDIR	if a component of the path prefix is not a directory
ENOTDIR	if an attempt is made to set a default ACL on a file type other than a directory
ENOENT	if a component of the pathname should exist but does not
EACCES	if the DAC and/or MAC check fails
EINVAL	if <i>cmd</i> is not ACL_CNT , ACL_GET , or ACL_SET
EINVAL	if <i>cmd</i> is ACL_SET and the ACL entries do not pass the validation check
ENOSPC	if <i>cmd</i> is ACL_GET and the space required for the file's ACL entries exceeds <i>nentries</i>
ENOSPC	if <i>cmd</i> is ACL_SET and there is insufficient space in the file system to store the ACL
EINVAL	if the number of acl entries exceeds the value of aclmax
ENOSYS	if the file system type does not support ACLs, and additional entries are specified

A.2.4 *aclsort* Function

DESCRIPTION: The *aclsort* function will take as input a buffer containing ACL entries (including default ACL entries) and sort them into the correct order to be accepted by the *acl* or the *aclipc* function. It will optionally calculate the maximum permissions needed for the object group class and set the **class** ACL entry.

SYNOPSIS:

```
#include <tbh.h>
```

```
int aclsort(int nentries, int calclass, struct acl *aclbufp)
```

Where the value of *nentries* is the number of ACL entries, the value of *calclass* if non-zero indicates to recalculate the **class** entry, and *aclbufp* is a pointer to ACL entry structures.

PROCESSING: A call to *aclsort* will result in the contents of the buffer being sorted in the following order:

- a) a user entry for the object owner.
- b) additional user entries.
- c) a group entry for the object owning group.
- d) additional group entries.
- e) a class entry for the file group class.
- f) an entry for other.
- g) default user entry for the object owner.
- h) default additional user entries.
- i) default group entry for the object owning group.
- j) default additional group entries.
- k) default class entry for the file group class.
- l) default entry for other.

Classes a), c), e), and f) must each have exactly one entry, if not, see **ERRORS AND RETURNS**. Classes g), i), k), and l) must have zero or one entry, if not, see **ERRORS AND RETURNS**. Entries will be sorted in increasing order, by user ID in classes b) and h), and by group ID in classes d) and j). Following

sorting, a check will be performed to verify that no duplicate entries (more than one entry containing the same user ID or the same group ID) exist. If duplicate entries are found, see **ERRORS AND RETURNS**.

If there are no entries in classes b) and d), the function will set the permission field, *a_perm*, in the class entry e) to that of the group entry c). If there are entries in classes b) or d) and the *calclass* argument is non-zero, the function will set the permission field, *a_perm*, of the **class** entry to the maximum permission of the entries in the file group class. Otherwise, the **class** entry permissions will remain unchanged.

If there are no entries in classes h) and j), the function will set the permissions in the default class entry k) to that of the default entry i).

Upon success, *aclsort* will return the value 0.

ERRORS AND RETURNS: If the *aclsort* function is unsuccessful due to duplicate entries, the return value will be the position (entry number) of the first duplicate entry. If there is less than one **user** entry for the object owner, **group** entry for the object owning group, **class** entry for the file group class, or **other** entry specified, a value of -1 will be returned. If there is more than one **user** entry for the object owner, **group** entry for the object owning group, **class** entry for the file group class, or **other** entry specified, they will be treated as duplicate entries, and the return value will be the position of the duplicate entry.

If the *aclsort* function is unsuccessful for any other reason, a value of -1 will be returned.

A.2.5 *chmod* Function

DESCRIPTION: The *chmod* function supports the following functionality:

1. it allows a subject to change the file mode, including the permissions for the file owner class, the file group class, and the file other class of a file.

Note that the *chmod* command will not require any modifications.

SYNOPSIS: No change.

PROCESSING: Any permissions changes made with the *chmod* command or function will update the file permission bits. This includes changing the file owner ACL entry, the **class** ACL entry, and the **other** ACL entry if the corresponding group(s) of bits are changed by this call. Any additional ACL entries will not be affected. Note, the permissions granted by such additional entries are constrained by the file group class permission bits. If no additional **user** and no additional **group** entries exist, the file group class permission bits will also represent the permissions for the owning group of the file.

ERRORS AND RETURNS: No change.

OUTPUT: No change.

A.2.6 *chown* Function

DESCRIPTION: The *chown* function supports the following functionality:

1. it allows a subject to change the owner and/or group of a file.

Note that the *chown* system call/command and the *chgrp* command will not require any modifications.

SYNOPSIS: No change.

PROCESSING: When the owner of a file is changed, the result will be identical to the current behavior. If the owner is changed to a user ID, for which an additional **user** entry already exists in the ACL, the additional **user** entry is not changed but the **user** entry for the file owner will take precedence during an access check. When the group of a file is changed, the result will be identical to the current behavior. If the group is changed to a group ID, for which an additional **group** entry already exists in the ACL, the additional **group** entry is not changed but the **group** entry for the file owning group will take precedence during an access check (except in the case of multiple concurrent groups, where all group entries are given equal treatment).

ERRORS AND RETURNS: No change.

OUTPUT: No change.

EXAMPLES: The following examples illustrate the operation of the *chown* function. For each example, there is a "before" state showing the output of *getacl*, the *chown* function that is executed, and the "after" state output.

EXAMPLE 1:

BEFORE:

```
# file: file1
# owner: larry
# group: guest
user::rwx
group::r--
class:r--
other:---
```

CALL: chown(file1, lisa, demo)

AFTER:

```
# file: file1
# owner: lisa
# group: demo
user::rwx
group::r--
class:r--
other:---
```

EXAMPLE 2:

BEFORE:

```
# file: file2
# owner: larry
# group: guest
user::rwx
user:fred:r--
group::r--
group:dev:r--
class:r--
other:---
```

CALL: chown(file2, lisa, demo)

AFTER:

```
# file: file2
# owner: lisa
# group: demo
user::rwx
user:fred:r--
group::r--
group:dev:r--
class:r--
other:---
```


EXAMPLE 3:

BEFORE:

```
# file: file3
# owner: larry
# group: guest
user::rwx
user:lisa:r--
user:fred:r--
group::r--
group:dev:r--
group:demo:r--
class:r--
other:---
```

CALL: chown(file3, lisa, demo)

AFTER:

```
# file: file3
# owner: lisa
# group: demo
user::rwx
user:lisa:r--
user:fred:r--
group::r--
group:dev:r--
group:demo:r--
class:r--
other:---
```

Note in EXAMPLE 3, a **user** entry contains a user ID that is the same as the file owner. In this case the file owner entry takes precedence. Also in EXAMPLE 3, a **group** entry contains a group ID that is the same as the owning group of the file. If multiple concurrent groups are not being used the object owning group entry takes precedence.

A.2.7 *aclipc* Function

DESCRIPTION: The *aclipc* call will support the getting and setting of discretionary permission information associated with an IPC object. It will allow the object owner or a process with appropriate permission or appropriate privilege to perform the following functions:

1. get or set an IPC object's ACL information in an atomic operation.
2. return the number of entries contained in an IPC object's ACL.

Note that this call only supports the IPC objects: e.g., shared memory segments, semaphores, and message queues. For simplicity, these objects are referred to as "IPC objects" in the remainder of this description.

SYNOPSIS:

```
#include <tbld.h>
```

```
int aclipc(int type, int id, int cmd, int nentries, struct acl *aclbufp)
```

Three values for *type* will be supported: **IPC_SHM**, **IPC_SEM**, and **IPC_MSG**. If *type* is **IPC_SHM**, *id* must be a valid shmid returned by *shmget*. If *type* is **IPC_SEM**, *id* must be a valid semid returned by *semget*. If *type* is **IPC_MSG**, *id* must be a valid msgid returned by *msgget*. Three values for *cmd* will be supported: **ACL_SET**, **ACL_GET**, and **ACL_CNT**. The value of *nentries* is the number of ACL entries that can fit in the user-supplied ACL buffer for an **ACL_GET** or the number actually present for an **ACL_SET**; and *aclbufp* is a pointer to the user-supplied buffer of ACL entry structures. The buffer will consist of an array of four (**USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, and **OTHER_OBJ** entries are required) or more occurrences of the following structure:

```
struct acl {  
    int a_type;  
    uid_t a_id;  
    ushort a_perm;  
};
```

Six values of *a_type* will be supported to specify the type of entry: **USER_OBJ**, **USER**, **GROUP_OBJ**, **GROUP**, **CLASS_OBJ**, and **OTHER_OBJ**. When *a_type* is **USER**, *a_id* will be a user id, and when *a_type* is **GROUP**, *a_id* will be a group id. When *a_type* is **USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, or **OTHER_OBJ**, *a_id* will not be used. The permissions for the entry will be contained in *a_perm*.

PROCESSING: When the specified *cmd* is **ACL_CNT**, the return value from the call will be the number of ACL entries for the IPC object specified by *type* and *id*. The values of *nentries* and *aclbufp* will be ignored. If the invoking user does not pass the DAC or MAC checks to see the ACL, the *aclipc* call will fail (see **ERRORS AND RETURNS**).

When the specified *cmd* is **ACL_GET**, the ACL information for the IPC object specified by *type* and *id* will be retrieved and the ACL entries will be placed in the buffer pointed to by *aclbufp*. The value of *nentries* is the number of entries that can be held in the buffer. If the number of ACL entries in the ACL is greater than the value of *nentries* (the buffer space allocated to hold the file's ACL entries is less than *nentries* times the size of an entry), the *aclipc* call will fail (see **ERRORS AND RETURNS**). On success, the return value from this call will be the number of ACL entries retrieved. On any error, the contents of the **acl** structures pointed to by *aclbufp* are indeterminate. If the user does not pass the DAC and MAC checks to see the ACL, the *aclipc* call will fail (see **ERRORS AND RETURNS**).

When the specified *cmd* is **ACL_SET**, ACL entries currently in the buffer, pointed to by *aclbufp*, for the IPC object specified by *type* and *id*, will be set if all required checks are passed. The contents of *nentries* shall be the number of ACL entries in the buffer pointed to by *aclbufp* to be copied. On success, the return value from this call will be 0. If the invoking subject does not pass the DAC and MAC checks to set an ACL, the *aclipc* call will fail (see **ERRORS AND RETURNS**). If an error occurs, either due to DAC or MAC checks or the validation check listed below, there will be no change to the current ACL information. Before the ACL entries are actually set, validation checks will be performed to determine that the ACL entries are in the following order:

- a) a user entry for the IPC object owner (**USER_OBJ**),
- b) additional user entries (**USER**),
- c) a group entry for the IPC object owning group (**GROUP_OBJ**),
- d) additional group entries (**GROUP**),
- e) a class entry for the IPC group class (**CLASS_OBJ**),
- f) an entry for other (**OTHER_OBJ**).

The entries in class a), c), e), and f) must always exist. The entry for class a), c), e), and f) do not use the *a_id* field. Class b) may contain zero or more

entries and the entries must be sorted by **uid** (lowest to highest). Class **d** may contain zero or more entries and the entries must be sorted by **gid** (lowest to highest). (this ordering should be done with the *aclsort* function).

Validation of the ACL will be performed. If entries containing duplicate *uids* or *gids* are found, or there is not exactly; one **user** entry for the object owner, one **group** entry for the object owning group, one **class** entry for the IPC group class, or one **other** entry specified, or there are no additional **user** and **group** entries and the permissions of the **class** entry are not equal to the permissions of the **group** entry, the call will fail (see **ERRORS AND RETURNS**).

The IPC owner permission bits will be changed, such that they are equal to the permissions specified for the **user** entry of the object owner in the ACL. The IPC group class permission bits will be changed, such that they are equal to the permissions specified for the **class** ACL entry. The IPC other class permission bits will be changed, such that they are equal to the permissions specified for the **other** ACL entry.

A design may constrain the maximum number of ACL entries that are written, with a system-wide tunable parameter, **aclmax**. If the number of ACL entries exceeds the value of **aclmax** the function will fail (see **ERRORS AND RETURNS**).

ERRORS AND RETURNS: If the *aclipc* call is unsuccessful, a value of **-1** will be returned and *errno* will be set to indicate the error. Only implementation-independent *errnos* are presented.

Under the following conditions, the function *aclipc* will fail and will set *errno* to the specified value (note: if *cmd* is unspecified, the *errno* applies to **ACL_CNT**, **ACL_GET**, and **ACL_SET**):

- | | |
|---------------|--|
| EINVAL | if <i>type</i> is not IPC_SHM , IPC_SEM , or IPC_MSG |
| EINVAL | if the value of <i>id</i> is (1) not a valid <i>message_queue_identifier</i> and the <i>type</i> was IPC_MSG , (2) not a valid <i>semaphore_identifier</i> and the <i>type</i> was IPC_SEM , or (3) not a valid <i>shared_memory_identifier</i> and the <i>type</i> was IPC_SHM |
| EINVAL | if <i>cmd</i> is not ACL_CNT , ACL_GET , or ACL_SET |

EINVAL	if <i>cmd</i> is ACL_SET and the ACL entries do not pass the validation check
EACCES	if the DAC and/or MAC check fails
ENOSPC	if <i>cmd</i> is ACL_GET and the space required for the IPC's object ACL entries exceeds <i>nentries</i>
ENOMEM	if <i>cmd</i> is ACL_SET and there is insufficient space to store the ACL
EINVAL	if the number of acl entries exceeds the value of aclmax

A.2.8 *shmctl*, *semctl*, & *msgctl* Functions

DESCRIPTION: The *shmctl*, *semctl*, and *msgctl* functions support the following functionality:

1. they allow a subject to change the user ID, group ID, and permissions on IPC objects.

SYNOPSIS: No change.

PROCESSING: No change.

ERRORS AND RETURNS: No change.

REFERENCES

- [1] *Department of Defense, Trusted Computer Systems Evaluation Criteria*, DoD 5200.28-STD, December 1985.
- [2] *National Computer Security Center, A Guide to Understanding Discretionary Access Control in Trusted Systems*, NCSC-TG-003 Version-1, September 1987.
- [3] *UNIX System Access Control List Proposal*, C. Rubin, AT&T, May 15, 1988.
- [4] *Adding Access Control Lists To UNIX*, A. Silverstein, B. McMahon, G. Nuss, Hewlett-Packard Co., March 12, 1988.
- [5] *Discretionary Access Control System Functions*, D. H. Steves, IBM, March 14, 1988.
- [6] *PI003.6 Security Extension Proposal: Discretionary Access Control Semantics*, W. Olin Sibert, Oxford Systems Inc., May 18, 1988.
- [7] *PI003.6 Supplementary Document: Discretionary Access Control: Problems in PI003.1 Draft 12*, W. Olin Sibert, Oxford Systems Inc., May 18, 1988.
- [8] *PI003.6 Supplementary Document: Comments on Hewlett-Packard ACL Proposal*, W. Olin Sibert, Oxford Systems Inc., May 18, 1988.
- [9] *Extending The UNIX Protection Model with Access Control Lists*, G. Fernandez, L. Allen, Apollo Computer Inc., June 1988.
- [10] *On Incorporating Access Control Lists into the UNIX Operating System* S. M. Kramer, SecureWare Inc., June 1988.
- [11] *Trusted UNIX Discretionary Access and Privilege Control Mechanisms*, B.D. Wilner, Infosystems Technology Inc., June 2, 1988.
- [12] *Access Control List Design*, Hewlett Packard, October 21, 1988.
- [13] *Proposal for Adding Access Control Lists to POSIX*, P. B. Flinn, SecureWare Inc., July 25, 1988.
- [14] *Discretionary Access Control Proposal*, H. L. Hall, Digital Equipment Corporation, Oct. 1988.
- [15] *Portable Operating System Interface for Computer Environments IEEE Std. 1003.1-1988*